ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

INGENIERÍA INFORMÁTICA

DISEÑO Y SIMULACIÓN DE UN PROCESADOR CUÁNTICO

Realizado por
JAIME Mª COELLO DE PORTUGAL VÁZQUEZ

Dirigido por
JOSÉ LUIS GUISADO LIZAR

Departamento
ARQUITECTURA Y TECNOLOGÍA DE COMPUTADORES

Sevilla, Junio de 2013

Índice

1 Introducción

1.1 Motivación

La miniaturización de los componentes que conforman los dispositivos electrónicos actuales se acerca cada vez más a un límite físico infranqueable: llegará un momento en el que los componentes serán tan pequeños que los efectos de la física cuántica serán más relevantes que los de la física clásica y nos será imposible manejar la información tal y como lo hacemos hasta ahora.

El efecto túnel, por ejemplo, permitiría a una corriente de electrones saltar de un conductor a otro aun estando separados por una barrera clásicamente infranqueable, lo que haría muy complicado manejarla.

Este límite nos obliga a buscar otras vías de avanzar en la implementación de computadores cada vez más rápidos. La idea básica de la computación cuántica es, en vez de ver la física cuántica como un límite, utilizar sus, a menudo contraintuitivas propiedades, para construir máquinas más potentes que las actuales.

Efectos como la superposición de estados cuánticos, que permite a una partícula estar en varios estados al mismo tiempo; o el entrelazamiento cuántico, que liga las partículas por muy separadas que estén, permite realizar computaciones y comunicar información de forma exponencialmente más rápida y totalmente segura.

Por supuesto todo esto dista de ser sencillo. Los estados cuánticos son extraordinariamente frágiles, tienden a interaccionar con su entorno de forma que se pierden sus características cuánticas para pasar a ser estados clásicos cuyas propiedades nos dejan de ser útiles. Por esto no observamos en el día a día los efectos microscópicos de las partículas, aun siendo todo un gran conjunto de ellas.

A día de hoy, la computación cuántica está en un estado muy temprano de su desarrollo, ya que se trata de un paradigma de la computación muy joven.

Las primeras ideas de utilizar las propiedades de la física cuántica para realizar computaciones que superasen a las clásicas las planteó Richard Feynman en 1982, al observar lo difícil que parecía para los ordenadores de su época simular sistemas cuánticos. Feynman postuló que un ordenador que utilizara las leyes de la física cuántica para funcionar, sería capaz de simular eficientemente sistemas cuánticos. Este postulado está afirmando, al fin y al cabo, que un ordenador tal y como los conocemos hoy en día

sería incapaz de realizar eficientemente ciertas tareas que un computador cuántico realizaría con facilidad.

En los años 80, David Deutsch, desarrolló el marco matemático actual de la computación cuántica, expuso el concepto de un computador cuántico universal y demostró con un sencillo ejemplo que podría realizar ciertas tareas más rápidamente que cualquier ordenador clásico, utilizando el algoritmo que lleva su nombre. Este algoritmo se explicará en detalla en las secciones siguientes.

En aquel momento la computación cuántica era un paradigma puramente científico, lejos de tener una aplicación práctica más allá de la simulación de otros sistemas cuánticos. Esta idea cambió radicalmente cuando Peter Shor propuso su algoritmo de descomposición de números primos. Apoyándose en el algoritmo de Deutsch-Josza, que permite obtener el periodo de una función, propuso un algoritmo para descomponer en factores primos, que supondría una mejora de orden exponencial sobre el mejor algoritmo clásico.

Dado que el extendido sistema de criptografía RSA se apoya en la dificultad que supone descomponer un número muy grande en sus factores primos, este descubrimiento supone, si se desarrollara un computador cuántico totalmente funcional, la caída de este sistema de criptografía.

Posteriormente se han ido descubriendo muchas otras aplicaciones de la información cuántica, hasta el punto que se ha dividido en varias ramas de desarrollo:

- Computación cuántica: el desarrollo de computaciones que superen de alguna forma a sus contrapartidas clásicas. Tanto el desarrollo teórico de *algoritmos cuánticos*, como la implementación física de computadores cuánticos propiamente dichos. Es en este punto en el que se centrará el proyecto. Hasta la fecha, el número de *bits cuánticos* que somos capaces de controlar en laboratorio es del orden de diez [4], muy lejos de lo necesario para superar a los sistemas actuales.
- Criptografía cuántica: el desarrollo de sistemas que permitan el envío de información de forma totalmente segura utilizando las propiedades de la física cuántica. El ejemplo clave de este tipo de algoritmos es el protocolo BB84 [20]. Dispositivos físicos de este tipo ya se comercializan, aunque tienen muchas limitaciones.
- Teleportación cuántica: el envío de información de forma más rápida haciendo uso del entrelazamiento cuántico. Teóricamente, se puede enviar la cantidad infinita de información contenida en el estado de una partícula de forma instantánea entre dos puntos del espacio, da igual lo distantes que estén. Este punto lleva a

confusión en muchos casos: primero, es información lo que se envía no materia; segundo, requiere el envío de dos bits clásicos, luego la información no viaja en ningún caso más rápido que la luz. En el momento de escribir esto, el record de distancia es de 143km entre la Palma y Tenerife [3].

Aunque no exista aun la tecnología para desarrollar un computador cuántico que supere a los sistemas de información actuales, el aspecto matemático de la computación cuántica ha sido desarrollado en detalle. Se tiene constancia de que, si bien un computador cuántico podría realizar cualquier operación que realizara uno clásico, es más lógico que un ordenador actual controle a la máquina cuántica. La gran ventaja es que podemos hacer un sistema de computación cuántica mucho más específico, lo justo para que la máquina conjunta sea totalmente universal en el sentido descrito por Deutsch, facilitando en gran medida su construcción.

Este es el punto fundamental de este proyecto: integrar en una arquitectura real de un procesador actual un núcleo que permita realizar una serie de operaciones sobre estados cuánticos. Este núcleo será, por supuesto, una simulación clásica de un estado cuántico.

En las siguientes secciones se detallara la construcción de este sistema y se dará pequeña introducción matemática a la información cuántica, de forma que sea más fácil comprender el resto del proyecto.

1.2 Objetivos

El objetivo general del proyecto es diseñar y simular en Java un procesador real, con la característica de que tendrá una unidad funcional capaz de ejecutar instrucciones de computación cuántica.

Este conjunto de instrucciones debe ser suficiente para ejecutar cualquier algoritmo cuántico que se desee, en más o menos tiempo, es decir, debe ser una máquina cuántica universal en la que se pueda ejecutar cualquier operación combinando las instrucciones implementadas.

El simulador desarrollado, por tanto, debe tener las siguientes propiedades:

- Ser capaz de simular los estados cuánticos de una forma lo más óptima posible, de forma que sea posible utilizar una cantidad de qubits razonable para la ejecución de una serie de algoritmos de prueba, que serán:
    - El algoritmo de Deutsch, demostración de la eficiencia del computador.
    - El algoritmo de Grover, de búsqueda en una base de datos desordenada.

- No debe ser simplemente un intérprete de instrucciones de un código maquina extendido, sino un simulador del procesamiento de las instrucciones por un computador real, con una arquitectura bien definida y realista, al menos en los componentes clásicos del procesador. En cada ciclo de ejecución las instrucciones deben modificar el estado interno del procesador (registros y memoria) y ejecutarse en una serie de dispositivos combinacionales simulados, tal y como lo harían en un procesador real

- Para ello imitará a los lenguajes de descripción de hardware, como VHDL, lo que permitirá diseñar el computador componente a componente de acuerdo a su arquitectura.

- Con respecto a la unidad funcional cuántica, se deben hacer las mínimas asunciones posibles con respecto a su funcionamiento, ya que se trata de una unidad ficticia, irrealizable con la tecnología actual, y desconocemos cómo se comportará cuando se desarrolle.

Por último se presentará una interfaz gráfica que permita hacer un seguimiento del estado del procesador en cada componente relevante, incluyendo el estado cuántico de la nueva unidad funcional.

1.3 Estructura de la memoria

En la sección **2: "Antecedentes"**, se da una introducción teórica a los conocimientos necesarios para comprender el resto del proyecto. El apartado **2.1: "Implementación física de un computador cuántico"**, da una visión de los hitos históricos y el estado actual de desarrollo de los computadores cuánticos reales. En el apartado **2.2: "La unión entre el procesador cuántico"** se explica por qué es importante esta unión y se da algún ejemplo de implementación real de ella. En **2.3: "Conocimientos necesarios"**, se da una introducción teórica básica a la física cuántica y se describe el procesador MIPS I real.

En la sección **3: "Desarrollo del proyecto"** se detalla la ejecución del proyecto en detalle. En su subapartado **3.1: "Simulando los sistemas cuánticos"** se explica cómo se van a representar e implementar los estados cuánticos en el simulador. En **3.2: "El simulador de circuitos cuánticos Qubit101"**, se describe este simulador a nivel de implementación y como está desarrollado en Java el motor de simulación de estados cuánticos. En **3.3: "Simulando hardware en Java"** se detalla cómo se imita a los lenguajes y simuladores de hardware en el proyecto. En **3.4: "Diseñando el procesador qMIPS"**, se describe en detalle la arquitectura e implementación de este procesador *clásico-cuántico*, así como de qué forma se integra la unidad cuántica en el sistema.

En la sección **4: "Manuales de uso"**, se explica el funcionamiento básico de ambas herramientas de forma que sean fáciles de utilizar por los usuarios.

La sección **5: "Implementación de algoritmos cuánticos"**, muestra el funcionamiento del simulador qMIPS utilizando como ejemplos el algoritmo de Deutsch y el de Grover.

En la sección **6: "Análisis temporal y de costes de desarrollo"**, se divide el proyecto en una serie de subtareas y se de una estimación del esfuerzo necesario para realizarlas. Finalmente se obtiene una estimación del coste total del proyecto.

En la sección **7: "Conclusiones"**, se exponen las conclusiones del proyecto, así como las posibles utilidades de la herramienta.

En la sección **8: "Futuras líneas"**, se marcan una serie de posibles mejorar para la herramienta de simulación.

Por último, la sección **9: "Bibliografía"**, recoge la documentación, literatura y direcciones web utilizadas para desarrollar el proyecto.

Existe un **Anexo A** que incluye el código fuente principal de las aplicaciones.

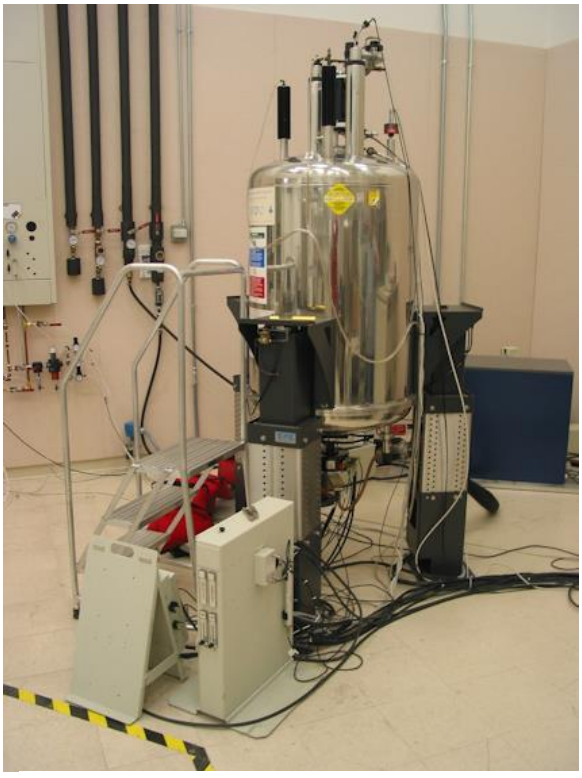
2 Antecedentes

2.1 Implementación física de un computador cuántico

El verdadero reto existente hoy en día en este campo es desarrollar un computador cuántico que sea capaz de mantener los estados totalmente aislados de forma que no pierdan sus propiedades cuánticas al interaccionar con su entorno, es decir, con unos tiempos de *decoherencia* suficientes para operar y con una tecnología que permita incrementar el número de qubits de forma arbitraria, es decir, que sea escalable.

Desarrollar un computador cuántico para computaciones pequeñas es relativamente fácil. Un espectrómetro de resonancia magnética nuclear (RMN), disponible en muchos laboratorios, se puede utilizar como un pequeño computador cuántico. La característica diferenciadora de este tipo de implementación es que la computación se ejecuta sobre una muestra con un gran conjunto de moléculas y no sobre una en particular.

En el 2001 un equipo de IBM realizo con éxito la factorización del número 15 en 3 y 5 mediante el algoritmo de Shor [18], utilizando un espectrómetro RMN sobre una muestra que contenía moléculas sintéticas, con cinco núcleos de ^{19}F y dos de ^{13}C. El espectrómetro utilizaba el espín de los núcleos como qubits, luego la máquina disponía de 7 qubits para operar.

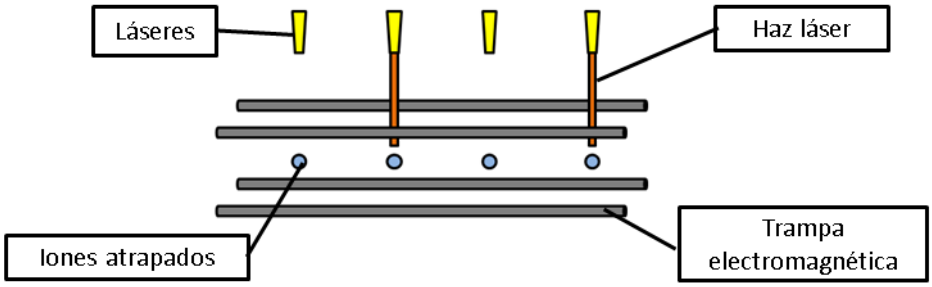


Espectrómetro RMN del mismo modelo que el utilizado por IBM. [http://www.mckscientific.com/]

Esta implementación es fácil de realizar y tiene tiempos de *decoherencia* relativamente largos. Aun así, no es un buen candidato para implementar un computador cuántico que supere a los ordenadores actuales ya que es tremendamente complicado aumentar el número de qubits disponibles.

Otra rama de avance en la implementación física de este tipo de máquinas se basa en lo que se denomina “trampas de iones”.

Las trampas de iones utilizan una combinación de campos electromagnéticos para confinar iones individuales en una región del espacio:



En 1995 J. I. Cirac y P. Zoller de la Universidad de Innsbruck, propusieron un modelo de computador cuántico utilizando trampas de iones que era capaz de realizar la operación “NOT-controlada” (se describirá más adelante) [19]. Para realizar las computaciones, los iones se enfrían hasta su estado fundamental utilizando un haz láser de frecuencia adecuada. Los qubits son los estados energéticos de los iones atrapados, luego tendremos un qubit por cada uno. Un haz laser por cada ion se encarga de operar con ellos, excitándolos o enfriándolos según sea necesario. En el modelo de Cirac y Zoller los iones que interaccionan no tienen por qué estar juntos.

En mayo de 2011 un grupo de investigadores de varias universidades publicó un artículo [4] en el que describían un método para mantener entrelazados, es decir, mantenidos en un estado cuántico coherente, hasta 14 iones en una trampa. Hasta la fecha es el mayor número de qubits mantenidos en una trampa iones.

Existen varios modelos más hoy día, muchos de ellos de una enorme complejidad, sin bien ninguno se acerca aún al rendimiento de un ordenador moderno. Remito a la bibliografía para obtener más información [1, 10, 12].

2.2 La unión entre el procesador cuántico y el clásico

El concepto de unión entre computador clásico y cuántico, concepto clave de este proyecto, está siendo explotado intensamente en el ámbito de la investigación. De hecho, se han realizado experimentos reales de computadores cuánticos a muy pequeña escala que combinan de forma efectiva ambos paradigmas.

Investigadores de la Universidad de California desarrollaron en el 2011 un concepto similar al simulador de este proyecto, si bien desde una perspectiva distinta: una implementación física de un computador cuántico de dos qubits cuya arquitectura este basada en la de Von Neumann, pilar principal de la mayoría de los computadores clásicos modernos [5]. Este computador disponía de una unidad de procesamiento cuántico con dos qubits unidos por un bus de acoplamiento, una memoria cuántica de otros dos qubits y dos registros de puesta a cero, todo ello integrado en un chip superconductor. La ventaja de esta arquitectura es que podemos pasar el estado de los qubits a la memoria cuántica donde tienen tiempos de decoherencia mucho más altos (sobre 1µs) que en los qubits sobre los que se opera (sobre 400ns), de esta forma se pueden almacenar en memoria mientras se realizan otras operaciones en la unidad de procesamiento. El programa a ejecutar sobre los qubits está almacenado en un ordenador corriente, que emite los pulsos de microondas necesarios.

Hoy en día es prácticamente unánime en la comunidad científica que este es el camino a seguir. Dado que el sistema cuántico solo es superior en escenarios muy específicos, es más lógico utilizar un ordenador corriente, mucho más fácil de desarrollar y programar, para realizar la mayoría de las tareas. El computador cuántico entraría en acción a petición del clásico, cuando se dé una situación para la que este es más efectivo.

2.3 Conocimientos necesarios

Para la correcta comprensión del Proyecto que se va a exponer, es necesario introducir al lector en el campo de la Física Cuántica. Se enfocará claramente hacia la rama de la Información Cuántica y de forma resumida, el lector interesado encontrará una extensa guía en [1]. Además, se presentará la arquitectura del procesador clásico MIPS, en la que se apoya el Proyecto [7].

2.3.1 Física e Información cuántica

La física cuántica no es más que un marco matemático para el desarrollo de teorías físicas. Se basa en una serie de postulados empíricos, obtenidos prácticamente por ensayo y error, que aun así han resultado en una importantísima rama de la física de una precisión impresionante, con tan solo algunos problemas que se han ido refinando en sucesivas teorías. Aquí no necesitaremos tal nivel de precisión y nos apoyaremos en el marco matemático clásico de la Física Cuántica.

Los postulados de la Física Cuántica, que definen dicho marco matemático, cambian dependiendo de la fuente que se consulte y de a qué rama se enfoque dicha fuente. Por supuesto todos vienen a decir lo mismo solo que planteado de diversas formas. Dado que aquí buscamos el punto de vista de la Información Cuántica, expondré dichos postulados citando a [1]. Me apoyaré en ellos para explicar los conceptos que sean necesarios para la comprensión del Proyecto. Una guía para iniciar a los Ingenieros Informáticos en el mundo de la computación cuántica se puede encontrar en [11].

Los postulados de la física cuántica

- **Primer postulado: El espacio de estados**

"Asociado a cualquier sistema físico aislado existe un espacio vectorial complejo con un producto interno definido (es decir, un espacio de Hilbert) denominado ***espacio de estados*** *del sistema. El sistema queda completamente descrito por su* ***vector de estado****, que es un vector unitario en el espacio de estados del sistema"*

-[1]

Como se puede observar la definición de este *espacio de estado* y su *vector de estados* deja mucho sin fijar. ¿Cuál es el espacio de estados de un sistema físico dado? La pregunta dista mucho de ser trivial, muchos espacios de estados de sistemas físicos estudiados hoy día son de una complejidad tremenda.

Afortunadamente, nosotros solo necesitamos el espacio de estados más simple que se puede plantear, lo que se denomina un **qubit.** Un qubit tiene un espacio de estados de tan solo dos dimensiones complejas. Tomando como base de dicho espacio dos vectores ortogonales, llamémosles $|0\rangle$ y $|1\rangle$, cualquier vector queda definido como:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

Con α y β dos números complejos de forma que el vector sea unitario, es decir $|\alpha|^2 + |\beta|^2 = 1$.

Esta notación se denomina *notación de Dirac*, los estados se representan con símbolos del tipo $|\phi\rangle$ (*ket*) y se pueden identificar con *vectores columna*, cada uno de ellos tiene asociado un vector dual, con el símbolo $\langle\phi|$ (*bra*), que se pueden identificar, a su vez, con vectores fila. Uno se obtiene a partir del otro transponiéndolo y complejo-conjugándolo (conjugación hermítica). El producto escalar de dos vectores de este tipo se escribe de forma sencilla: $\langle\phi|\,\psi\rangle$.

Estos estados se pueden representar de forma matricial. Como tenemos un espacio de dos dimensiones complejas podemos tomar una base que defina este espacio de forma arbitraria, es decir, cualesquiera dos vectores complejos ortogonales y unitarios definirán una base válida.

Etiquetando las bases como $|0\rangle$ y $|1\rangle$ es fácil relacionarlas con los bits clásicos. El significado físico de estos estados dependerá de la implementación física correspondiente: el espín de los núcleos en un espectrómetro RMN, el estado de excitación de los iones en una trampa, etc. Pero los cálculos matemáticos relevantes son independientes de la implementación, luego podemos abstraernos de la implementación física y aun así proponer un marco matemático coherente para la computación cuántica.

Una base del espacio complejo de dos estados que simplifica los cálculos, al menos para esta introducción, es la que se denomina *base computacional*, definida como:

$$|0\rangle = \begin{pmatrix}1\\0\end{pmatrix}, |1\rangle = \begin{pmatrix}0\\1\end{pmatrix}$$

De forma que un vector arbitrario queda definido como:

$$|\psi\rangle = \begin{pmatrix}\alpha\\\beta\end{pmatrix}$$

Siempre y cuando $|\alpha|^2 + |\beta|^2 = 1$.

Para intentar entender las similitudes y diferencias entre las computaciones clásica y cuántica, podemos relacionar sus bases de la forma:

0 lógico = $|0\rangle$

1 lógico = $|1\rangle$

De esa forma podríamos operar con los qubits, a simple vista, como si de lógica binaria se tratase. Pero el estado de un qubit no se reduce simplemente a $|0\rangle$ o $|1\rangle$, existen estados que son perfectamente válidos como por ejemplo: $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$, de los cuales no podemos decir que estén en el estado $|0\rangle$ ni en el $|1\rangle$. La interpretación que se debe dar a este tipo de estados ni siquiera está clara hoy en día, se podría pensar que es la falta de información sobre el sistema, debida quizás a que la teoría cuántica es incompleta, la que nos está llevando a estados absurdos, a una simple representación probabilística de nuestro desconocimiento, es decir, la existencia de *variables ocultas* que no se están incluyendo en la teoría. Contra este enfoque, John Stewart Bell, publicó en 1964 un artículo [21] en el que, partiendo tan solo de dos premisas:

- Realidad: los sistemas físicos tienen valores definidos para sus propiedades aunque no haya nadie observándolos.
- Localidad: la información, es decir, cualquier efecto físico, se propaga a una velocidad finita.

Premisas que definen el mundo como lo conocemos macroscópicamente, obtuvo una desigualdad. Siguiendo el mismo camino, pero utilizando el marco matemático de la física cuántica, descubrió que dicha desigualdad se violaba, lo que quiere decir que si la física cuántica describe la naturaleza, la asunción de *realismo local* es falsa. En contra a lo que se podría pensar, experimentos llevados a cabo para poner a prueba estas desigualdades se han decantado siempre del lado de la física cuántica y la comunidad científica acepta de forma general la violación de las desigualdades de Bell, si bien no ha conseguido hasta el momento desarrollar ningún experimento para verificarlas que esté completamente libre de lagunas [14].

Por tanto, aquí asumiremos que los estados del tipo $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$ están representando que el nivel energético, espín, etc. se encuentra en ambos estados al mismo tiempo. Esto se denomina *superposición cuántica de estados*, y es una de las claves de la potencia de la computación cuántica.

El problema es que ese tipo de estado es inobservable. Como señalaré en el tercer postulado, el hecho de observar un estado cuántico, en general, lo destruye. Pero que no podamos observar dichos estados no significa, como ya se ha mostrado, que sean ficticios; podemos obtener, si bien no todos los resultados posibles de una función, sí una propiedad general de esta, como puede ser su periodo. Esto será fundamental para los algoritmos cuánticos más famosos, que superan ampliamente a las máquinas clásicas.

Por último, comentar la forma más utilizada para representar algoritmos cuánticos: los circuitos cuánticos. En un circuito cuántico un qubit se representa simplemente como una línea continua:

$|\psi\rangle =$

Es importante señalar que estos *circuitos* no son en ningun caso circuitos físicos como pueden ser los eléctricos, se trata simplemente de una representacion gráfica de un algoritmo que en su aplicación a un computador cuántico real se traducirian en una serie de pulsos laser, de radiofracuencia, etc; dependiendo de la implementación física de dicha máquina.

En los siguientes puntos se irá completando la descripción de este método para representar algoritmos cuánticos.

- **Segundo postulado: La evolución de los estados**

"La evolución de un sistema cuántico cerrado viene descrito por una transformación unitaria. Es decir, el estado $|\psi\rangle$ *del sistema en el instante* t_1 *está relacionado con el estado en el instante* t_2 *por un* ***operador unitario*** *U que depende únicamente de los instantes* t_1 *y* t_2*."*

-[1]

Esta evolución es una evolución discreta, nos lleva del instante t_1 al t_2 instantáneamente. Si queremos obtener cómo evoluciona el estado de forma continua tendremos que recurrir a la ecuación de Schrödinger pero por ahora no nos será necesaria.

Este postulado nos da intrínsecamente un elemento clave: podemos hacer evolucionar a los estados a voluntad, es decir, podemos operar con ellos siempre y cuando garanticemos que las operaciones que se realizan son unitarias.

Una vez más, cabe aclarar que esta forma matemática de esta evolución se puede definir sin necesidad de elegir una implementación física. A partir de esta forma matemática podremos, posteriormente, obtener las ecuaciones concretas para una implementación física dada. Pero esto dista mucho de ser trivial y escapa al alcance de este proyecto.

Cuando un operador de este tipo se aplica a un qubit se denomina *puerta cuántica.*

Un operador es unitario cuando cumple que:

$$U^{\dagger}U = UU^{\dagger} = I$$

Un ejemplo de una puerta cuántica muy útil sería la contrapartida cuántica del inversor clásico (puerta NOT), la puerta cuántica X:

$$NOT(0) = 1, NOT(1) = 0$$

$$X|0\rangle = |1\rangle, X|1\rangle = |0\rangle$$

En la base computacional:

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}; \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix};$$

Es fácil comprobar que es unitaria.

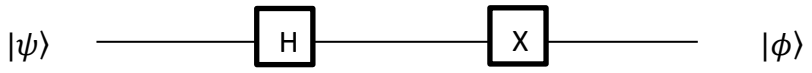
La puerta cuántica quizás más importante se denomina *puerta de Hadamard* y nos permite hacer evolucionar los estados base hacia una superposición cuántica, ya que realiza:

$$H|0\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle \text{ y } H|1\rangle = \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle$$

Su representación matricial es:

$$H = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

En la representación de circuitos cuánticos, las puertas se representan como cajas etiquetadas con el nombre de la puerta:



El tiempo pasa de izquerda a derecha de la linea. En el ejeplo quedaría que $|\phi\rangle = X(H|\psi\rangle)$.

- **Tercer postulado: Las medidas**

"Las medidas cuánticas vienen descritas por una colección $\{M_m\}$ de operadores de medida. Estos operadores actúan sobre el espacio del sistema bajo medida. El subíndice 'm' indica una de las posibles respuestas de la medida. Si el estado previo a la medida es el $|\psi\rangle$, la probabilidad de obtener un resultado 'm' será:

$$p(m) = \langle\psi|M_m^\dagger M_m|\psi\rangle.$$

y el estado tras la medida quedará:

$$\frac{M_m|\psi\rangle}{\sqrt{\langle\psi|M_m^\dagger M_m|\psi\rangle}}.\text{"}$$

-[1]

En el segundo postulado vimos como evolucionaba un estado cuántico en un sistema cerrado. Pero si queremos obtener resultados de los experimentos no tenemos más remedio que interferir en él. El hecho de observar un estado cuántico lo convierte en un sistema abierto y la evolución unitaria ya no es válida.

Los operadores de medida son proyectores sobre el subespacio sobre el que se realiza la medida; véase el subespacio definido por $|0\rangle$ para un resultado de 0 y $|1\rangle$ para un resultado de 1. Entonces vemos que, como era de esperar, si medimos el estado $|1\rangle$ es imposible obtener un 0 como resultado ya que su proyección sobre el subespacio 0 es nula. Esta proyección es la que da la probabilidad de obtener un resultado u otro.

Por ejemplo, si suponemos el estado $\frac{1}{\sqrt{3}}|0\rangle + \sqrt{\frac{2}{3}}|1\rangle$ y lo medimos obtendremos:

$$\begin{cases} 0, con\ probabilidad: \left(\langle 0|\frac{1}{\sqrt{3}} + \langle 1|\sqrt{\frac{2}{3}}\right) M_0^\dagger M_0 \left(\frac{1}{\sqrt{3}}|0\rangle + \sqrt{\frac{2}{3}}|1\rangle\right) = \left(\frac{1}{\sqrt{3}}\right)^2 = \frac{1}{3} \\ 1, con\ probabilidad: \left(\langle 0|\frac{1}{\sqrt{3}} + \langle 1|\sqrt{\frac{2}{3}}\right) M_1^\dagger M_1 \left(\frac{1}{\sqrt{3}}|0\rangle + \sqrt{\frac{2}{3}}|1\rangle\right) = \left(\sqrt{\frac{2}{3}}\right)^2 = \frac{2}{3} \end{cases}$$

Y tras la medida el estado quedará:

$$\begin{cases} \dfrac{\mathrm{M}_0\left(\frac{1}{\sqrt{3}}|0\rangle + \sqrt{\frac{2}{3}}|1\rangle\right)}{\sqrt{1/3}} = |0\rangle, si\ se\ obtubo\ un\ 0 \\ \dfrac{\mathrm{M}_1\left(\frac{1}{\sqrt{3}}|0\rangle + \sqrt{\frac{2}{3}}|1\rangle\right)}{\sqrt{2/3}} = |1\rangle, si\ se\ obtubo\ un\ 1 \end{cases}$$

Sabiendo que el proyector $\mathrm{M}_0 = |0\rangle\langle 0|$ y $\mathrm{M}_1 = |1\rangle\langle 1|$.

En resumen, el módulo al cuadrado de los coeficientes que acompañan a cada vector en un estado cuántico nos da la probabilidad de encontrarnos dicho estado tras una medida y por tanto de obtener su resultado asociado.

El símbolo asociado a las medidas en un circuito cuántico es: 

- **Cuarto postulado: los sistemas físicos compuestos**

"El espacio de estados de un sistema físico compuesto es el producto tensorial de los espacios de estados componentes. Es más, si tenemos los estados numerados de 1 a 'n' y el sistema 'i' se prepara en el estado $|\psi_i\rangle$, el estado conjunto del sistema completo será $|\psi_1\rangle \otimes |\psi_2\rangle \otimes \dots \otimes |\psi_n\rangle$."

-[1]

Hasta ahora habíamos trabajado con espacios tan solo de un qubit. ¿Cómo representamos estados en los que está involucrado más de un qubit? Realizando el producto tensorial de ambos estados. Por ejemplo si tenemos el estado $|\psi_1\rangle = |0\rangle$ y el $|\psi_2\rangle = |1\rangle$ el estado total resultante será: $|0\rangle \otimes |1\rangle$, o simplemente $|01\rangle$.

El producto de tensorial en su forma matricial se calcula de la siguiente forma, para un producto de un vector de dimensión dos por uno de cuatro:

$$\begin{pmatrix} a_1 \\ a_2 \end{pmatrix} \otimes \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} = \begin{pmatrix} a_1 \cdot \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} \\ a_2 \cdot \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} \end{pmatrix} = \begin{pmatrix} a_1 \cdot b_1 \\ a_1 \cdot b_2 \\ a_1 \cdot b_3 \\ a_1 \cdot b_4 \\ a_2 \cdot b_1 \\ a_2 \cdot b_2 \\ a_2 \cdot b_3 \\ a_2 \cdot b_4 \end{pmatrix}$$

Un sistema de dos qubits ya no está inmerso en un espacio complejo de dos dimensiones, sino de cuatro. En forma matricial podemos hacer (en la base computacional), por ejemplo:

$$|0\rangle \otimes |1\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix} \\ 0 \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix} \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} = |01\rangle$$

Como vemos los vectores que representan el estado de dos qubits tienen dimensión cuatro. En general, la dimensión del sistema completo será de 2^q, con 'q' el número de qubits.

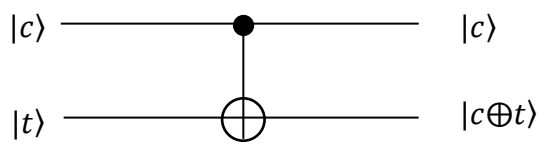
Por otra parte, las puertas cuánticas que afectan a un espacio de varios qubits ya no pueden ser de dimensión dos. Los operadores que operan sobre este espacio son el producto tensorial de los operadores que actúan sobre los subespacios correspondientes, por ejemplo, $X \otimes H$ aplica la puerta X al primer qubit y la puerta H al segundo. Nótese que la operación no es conmutativa, $H \otimes X$ aplica H al primer qubit y X al segundo y su representación matricial será distinta en general.

El producto tensorial de dos matrices se calcula, para una matriz 2x2 por otra 2x2, de la forma:

$$\begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix} \otimes \begin{pmatrix} b_1 & b_2 \\ b_3 & b_4 \end{pmatrix} = \begin{pmatrix} a_1 \cdot \begin{pmatrix} b_1 & b_2 \\ b_3 & b_4 \end{pmatrix} & a_2 \cdot \begin{pmatrix} b_1 & b_2 \\ b_3 & b_4 \end{pmatrix} \\ a_3 \cdot \begin{pmatrix} b_1 & b_2 \\ b_3 & b_4 \end{pmatrix} & a_4 \cdot \begin{pmatrix} b_1 & b_2 \\ b_3 & b_4 \end{pmatrix} \end{pmatrix}$$
$$= \begin{pmatrix} a_1 \cdot b_1 & a_1 \cdot b_2 & a_2 \cdot b_1 & a_2 \cdot b_2 \\ a_1 \cdot b_3 & a_1 \cdot b_4 & a_2 \cdot b_3 & a_2 \cdot b_4 \\ a_3 \cdot b_1 & a_3 \cdot b_2 & a_4 \cdot b_1 & a_4 \cdot b_2 \\ a_3 \cdot b_3 & a_3 \cdot b_4 & a_4 \cdot b_3 & a_4 \cdot b_4 \end{pmatrix}$$

En computación clásica, un conjunto de instrucciones de vital importancia son las condicionales, que ejecutan una operación solo si se cumple una determinada condición. En computación cuántica también existen este tipo de operaciones. La más simple y más utilizada es la puerta cuántica CNOT (*Controlled-NOT, Negación controlada*), que ejecuta la operación X (NOT) sobre un qubit denominado *objetivo* solo si otro denominado *control* está en el estado $|1\rangle$. Es decir, si tenemos un par de qubits $|c, t\rangle$, la operación CNOT realiza: $\mathrm{CNOT}|c, t\rangle = |c, c \oplus t\rangle$, siendo el operador $\oplus$ una suma módulo 2 o la operación XOR, ya que si c = 0 deja inalterado a *t* ($0 \oplus 0 = 0, 0 \oplus 1 = 1$) y si c = 1 lo invierte ($1 \oplus 0 = 1, 1 \oplus 1 = 0$).

Un circuito cuántico tiene una línea horizontal por cada qubit que se quiera representar. La puerta CNOT tiene una representación especial, un punto negro indica el qubit control y el símbolo $\oplus$ indica el qubit objetivo, ambos conectados por una línea:

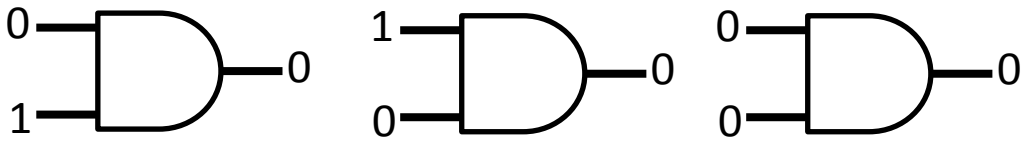


Detalles a la hora de construir algoritmos cuánticos

Dados los postulados vistos en el apartado anterior, encontramos una serie de restricciones para la construcción de *programas cuánticos*, la restricción más fuerte nos la da que nuestras puertas, que hacen evolucionar el estado, deben ser unitarias y por ende:

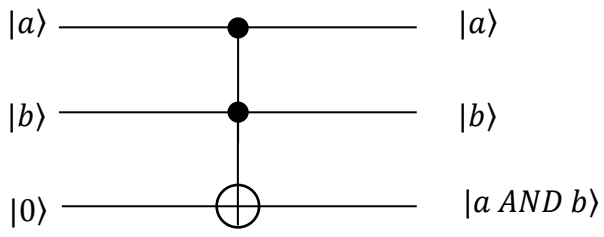
a) **Deben ser reversibles**: Al cumplir las puertas que $U^{\dagger}U = UU^{\dagger} = I$, lo que estamos afirmando es que su *conjugada hermítica* (†) es su inversa, es decir, su inversa siempre existe. Por tanto, siempre debemos ser capaces de reconstruir la entrada de una puerta a partir de su salida.

Si se observan algunas puertas clásicas, como por ejemplo la puerta AND, resulta que es imposible obtener la entrada a partir de la salida para ciertos casos:



Como se observa a partir de su salida 0 es imposible saber que entrada se introdujo.

Entonces, ¿no se puede realizar una puerta AND cuántica? Sí se puede construir, pero debemos conservar información suficiente para reconstruir la entrada y para ello no es suficiente con dos entradas para esta puerta en concreto:



Necesitamos, como en la figura, un qubit auxiliar.

Esta puerta reversible con dos qubits de control se conoce como *puerta de Toffoli*. Invierte el qubit inferior tan solo si los dos superiores valen 1. Introduciendo un estado $|1\rangle$ en el qubit inferior realiza la misma función que la operación *NAND*. Como la

operación *NAND* es universal para las puertas clásicas, queda demostrado que cualquier puerta clásica se puede ejecutar utilizando una combinación de puertas reversibles.

b) **Es imposible clonar un estado cuántico desconocido:** Lo que se conoce como *teorema de no-clonación*. La prueba es simple; imaginemos dos estados $|\psi\rangle$ y $|\varphi\rangle$ a copiar y una puerta C que realiza la copia:

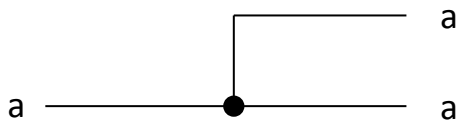
$$C(|\psi\rangle \otimes |s\rangle) = |\psi\rangle \otimes |\psi\rangle$$
$$C(|\varphi\rangle \otimes |s\rangle) = |\varphi\rangle \otimes |\varphi\rangle$$

Si realizamos el producto interno de estas dos ecuaciones obtenemos:

$$\langle\psi|\varphi\rangle = (\langle\psi|\varphi\rangle)^2$$

Pero la ecuación $x = x^2$ tan solo tiene dos soluciones, o x=1 o x=0. Por tanto los estados o son iguales o son ortogonales. Así que es imposible construir una puerta que clone un estado cuántico cualquiera.

Para los acostumbrados a la computación clásica, esto es un resultado impactante. Algo tan visto en cualquier circuito digital como:



No se puede realizar en una computación cuántica.

Con estos dos puntos, pensando en circuitos cuánticos, concluimos que ignorando puertas no unitarias como la de la medida, siempre tendremos a la salida el mismo numero de qubits que teniamos a la entrada.

El algoritmo de Deutsch

El primer algoritmo que se desarrolló que demostró que los computadores cuánticos pueden ser superiores a los clásicos fue desarrollado por David Deutsch en 1985 [6]. El algoritmo no tiene ninguna aplicación práctica más allá de demostrar esta superioridad.

El *problema de Deutsch* que resuelve este algoritmo es el siguiente: dado un *oráculo* que ejecuta una de las cuatro funciones binarias de un bit, determinar si se trata de una de las

dos funciones constantes: $f(x) = 0$ o $f(x) = 1$; o una de las funciones equilibradas: $f(x) = x$ o $f(x) = \bar{x}$.

Un *oráculo* es una función cuyo comportamiento interno es desconocido o simplemente no es importante (una caja negra). Algunos algoritmos cuánticos se basan en este tipo de funciones para operar.

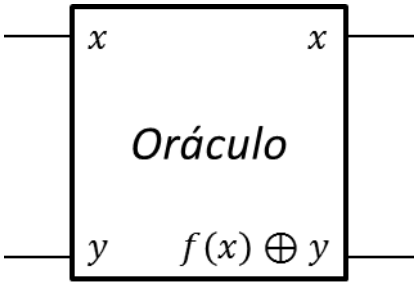
El algoritmo de Deutsch es capaz de resolver el problema con tan solo una llamada al *oráculo* contra el mínimo de dos que necesita un computador clásico.

Para ejecutar este algoritmo tan solo son necesarios dos qubits.

El estado de entrada al algoritmo debe ser el $|01\rangle$. El primer paso es aplicar una puerta de Hadamard a ambos qubits de forma que el estado quede:

$$\left[\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)\right]\left[\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)\right]$$

El oráculo se aplica negando el segundo qubit si el resultado de aplicar la función sobre el primero da 1. En forma de circuito:



De esta forma, el estado del sistema sería, tras la llamada al oráculo:

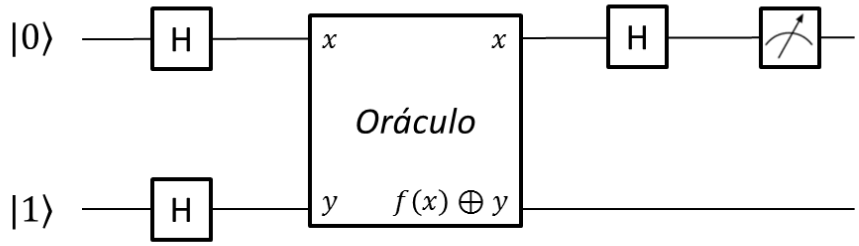
$$\begin{cases} \pm\left[\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)\right]\left[\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)\right] si\ f(0) = f(1) \\ \pm\left[\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)\right]\left[\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)\right] si\ f(0)\ \neq f(1) \end{cases}$$

Y si, por último, aplicamos una puerta de Hadamard al primer qubit:

$$\begin{cases} \pm|0\rangle\left[\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)\right] si\ f(0) = f(1) \\ \pm|1\rangle\left[\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)\right] si\ f(0)\ \neq f(1) \end{cases}$$

Ya es fácil observar que si medimos el primer qubit y obtenemos un 0, nos aseguramos de que $f(0) = f(1)$ y que por tanto la función $f(x)$ es de tipo constante. Sin embargo si obtenemos un 1, se tiene que $f(0) \neq f(1)$ y la función es equilibrada.

El circuito completo del algoritmo quedaría:

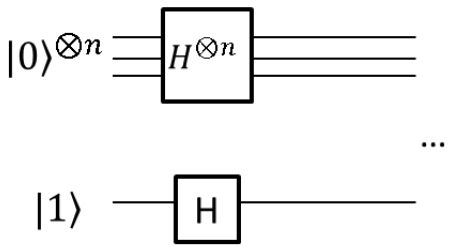


Este algoritmo al fin y al cabo está calculando una propiedad global de la función *f(x)*: su periodo, con tan solo una ejecución de dicha función. Esta capacidad para obtener propiedades globales de las funciones es un punto clave en muchos algoritmos cuánticos, como por ejemplo el algoritmo de factorización de Shor.

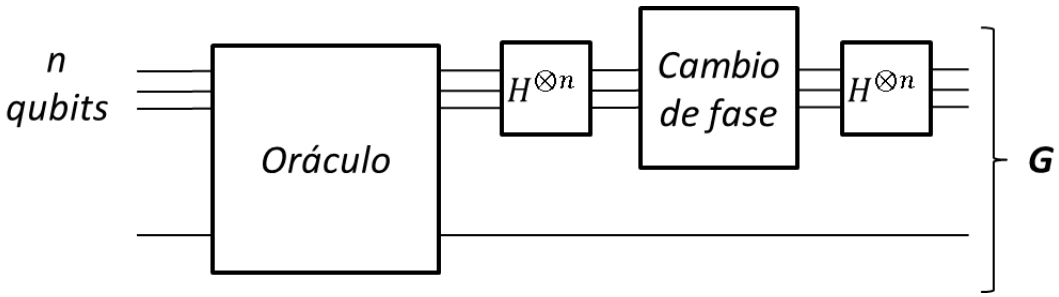
El algoritmo de Grover

Este algoritmo cuántico fue desarrollado por Lov K. Grover en 1996 [8]. Trata de buscar en una colección desordenada de datos uno o varios elementos que cumplan una cierta propiedad. Aquí particularizaré para el caso en el que tan solo exista una solución pero es sencillo generalizarlo para varias de ellas [1]. Este algoritmo es útil para aquellos problemas en los que identificar una solución es sencillo pero encontrarla es tremendamente complicado. Es capaz de encontrar dicha solución en un tiempo de orden $O(\sqrt{N})$, donde N es el número de elementos de la colección, contra el mejor algoritmo clásico que lo realizaría en un tiempo de orden $O(N)$. Se trata de un algoritmo que en general es probabilístico y su objetivo será intentar que la solución correcta sea la más probable.

El primer paso del algoritmo es colocar $n = \lceil \log(N) \rceil$ qubits en superposición utilizando puertas de Hadamard. Además, se debe utilizar un qubit más para el subespacio del oráculo preparado como en el algoritmo de Deutsch en el estado: $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$:



Ahora hay que repetir $O(\sqrt{N})$ veces el *operador de Grover*:

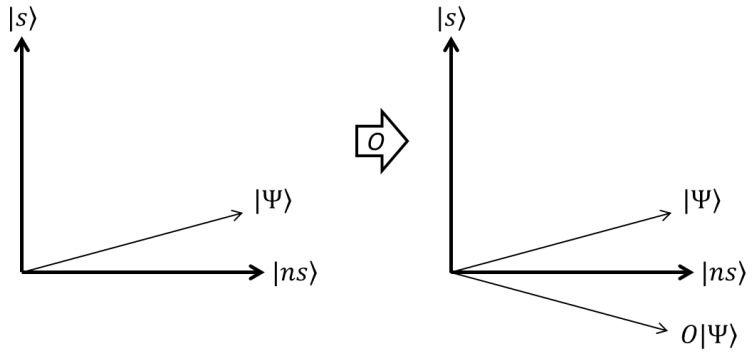


¿Cómo funciona este operador? Los *n* qubits superiores están inicialmente en el estado: $|\Psi\rangle = \frac{1}{\sqrt{N}}\sum_{x=0}^{N-1}|x\rangle$, así que, si $|s\rangle$ es el estado solución y $|ns\rangle$ es el estado conjunto del resto de componentes, podemos separarlo de la forma:

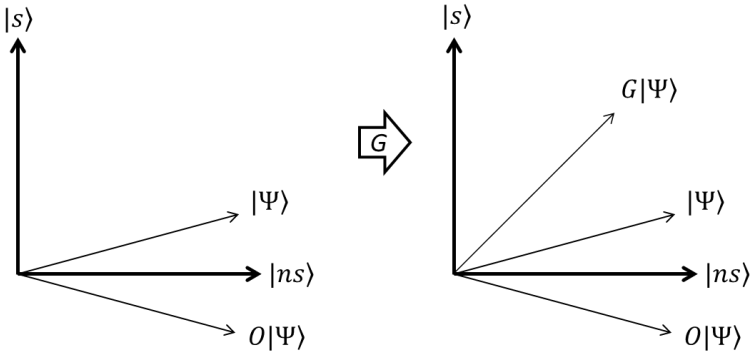
$$|\Psi\rangle = \sqrt{\frac{N-1}{N}}|ns\rangle + \sqrt{\frac{1}{N}}|s\rangle$$

Por simplicidad, $|\Psi\rangle = a|ns\rangle + b|s\rangle$.

Al estar el qubit auxiliar en el estado preparado, el efecto del oráculo será simplemente el de cambiar la fase de la componente solución: $a|ns\rangle - b|s\rangle$. Si visualizamos las componentes en su forma vectorial, podemos entender este efecto como una *reflexión* en el plano definido por $|ns\rangle$ y $|s\rangle$ sobre el vector $|ns\rangle$:

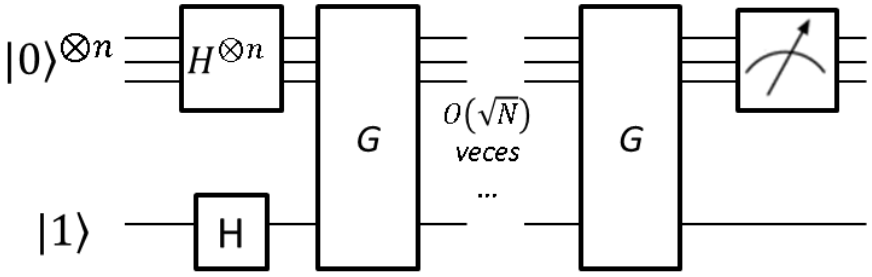


El siguiente paso del algoritmo es aplicar el operador $H(Cambio\ de\ fase)H$. El operador de cambio de fase invierte todas las componentes de la superposición excepto la componente $|0\rangle$, es decir, realiza la operación $2|0\rangle\langle 0| - I$. El efecto conjunto del operador sería $H^{\otimes n}(2|0\rangle\langle 0| - I)H^{\otimes n} = 2|\Psi\rangle\langle\Psi| - I$, que geométricamente es una vez más una reflexión, esta vez sobre el vector $|\Psi\rangle$:



Lo que hemos conseguido con esto es girar el vector $|\Psi\rangle$ para aproximarlo hacia $|s\rangle$, vector solución del problema. Es fácil ver que si repetimos estas reflexiones un número adecuado de veces, llegara un punto de acercamiento máximo al vector solución y, por tanto, una probabilidad máxima de obtener la respuesta correcta en la medida. Se puede demostrar que el número de veces que se debe aplicar el operador de Grover para garantizar la probabilidad máxima es $R \leq \left\lceil \frac{\pi}{4}\sqrt{N} \right\rceil$ [1], es decir, el orden de complejidad del algoritmo es $O(\sqrt{N})$.

El circuito completo del algoritmo quedaría:



Este algoritmo no sirve tan solo para buscar un dato en una base de datos desordenada, también puede ayudar a acelerar la solución de problemas de la clase NP. Por ejemplo, se puede utilizar para determinar si un grafo tiene un ciclo Hamiltoniano, si encontramos un oráculo que los identifique, podemos encontrarlos en un tiempo $O(\sqrt{N})$.

2.3.2 La arquitectura MIPS

La compañía “MIPS Computer Systems Inc.” nació en 1984 a partir de un proyecto de la Universidad de Stanford, para comercializar la arquitectura que el grupo MIPS CPU había desarrollado [7].

Se trataba de una arquitectura RISC con un encadenamiento estricto de 5 etapas. Estas dos características se combinan muy bien, ya que para poder garantizar un flujo constante de instrucciones todas deben ser del mismo tamaño y por tanto, no muy complejas ya que no hay suficiente espacio en la instrucción para complicar las instrucciones.

Una arquitectura RISC (*Reduced Instruction Set Computing*) es aquella que en contraposición a las CISC (*Complex Instruction Set Computing*) tiene un conjunto de instrucciones que tiende a ser muy simple, de forma que el hardware necesita muy pocos ciclos para ejecutar una instrucción completa y los compiladores pueden optimizar la ejecución de forma fácil ya que no suelen utilizar operaciones muy complejas y específicas que proporciona un procesador CISC. Este tipo de arquitectura ha tenido tanto éxito que ninguna otra desarrollada posteriormente al 1985 ha seguido otra filosofía. Los procesadores RISC no suelen permitir operar directamente con la memoria, sino que debemos cargar el dato, operar con él y guardarlo de nuevo, estas arquitecturas se denominan *load/store* y es el caso de la arquitectura MIPS.

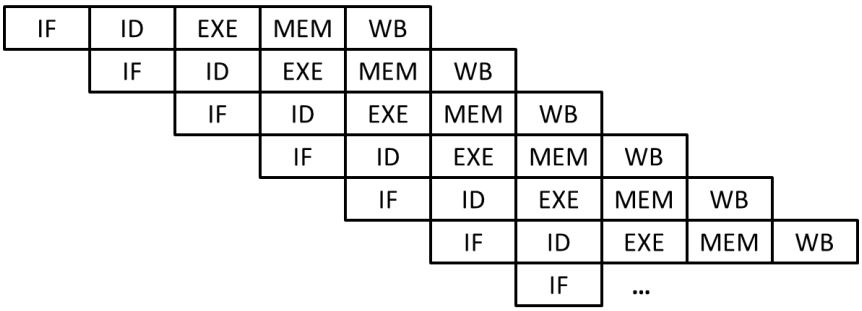
Los procesadores con encadenamiento dividen las instrucciones en una serie de pasos en cuya ejecución el hardware utilizado no se necesita en ningún otro paso, de forma que se pueden lanzar todas al mismo tiempo para acelerar la ejecución. El procesador MIPS I utilizaba una segmentación en 5 etapas:

- Fase IF (*Instruction Fetch*): La siguiente instrucción se obtiene de la memoria (o caché de instrucciones).
- Fase ID (*Instruction Decode*): Se decodifica la instrucción y se leen los operandos del fichero de registros.
- Fase EXE (*Execution*): Se utiliza la ALU para realizar los cálculos correspondientes ya sea una operación de tipo aritmético-lógica o el cálculo de una dirección efectiva en memoria.
- Fase MEM (*Memory*): Se realizan las operaciones con memoria (o caché de datos) ya sea lectura o escritura de los datos correspondientes. Aproximadamente tres de

cada cuatro instrucciones no harán nada en esta etapa, pero mantenerla evita que dos instrucciones intenten acceder a la memoria al mismo tiempo.

- Fase WB (*Write Back*): Se escribe en el registro correspondiente el resultado de la ejecución de la instrucción.

Y las ejecuta de la siguiente forma:



Se puede ver que tras las cinco primeras instrucciones y, en el caso ideal, el procesador ejecutaría cinco instrucciones al mismo tiempo en cada ciclo, finalizando una instrucción por ciclo. Es solo en el caso ideal porque las dependencias de datos (que una instrucción posterior necesite un dato de la anterior) o las dependencias de control (saltos y ramificaciones), provocan que hagan falta ciclos de bloqueo para que el programa se ejecute correctamente. Además, la idea de que cada fase dure exactamente un ciclo de reloj no es muy realista, ya que por ejemplo, buscar los datos en memoria que no estén en la caché o realizar operaciones de división o multiplicación puede ser demasiado costoso para ejecutarse en un ciclo.

Ahora podemos ver la justificación de que utilicemos una filosofía *load-store* ya que los datos se leen de memoria en la fase 4, donde ya es demasiado tarde para que se pueda operar con ellos en la ALU. Además las instrucciones RISC de un tamaño constante nos proporcionan un flujo más o menos constante en la fase IF.

El procesador del proyecto imita el conjunto de instrucciones del MIPS I con algunas variaciones, pero no es un procesador encadenado. Una ejecución de este tipo complica en gran manera la tarea ya bien del programador, del compilador o de la unidad de control, que deben detectar y prevenir los fallos por dependencias.

2.4 Alternativas

El campo de la simulación de circuitos cuánticos está actualmente bastante desarrollado. Una búsqueda simple nos permite obtener muchas alternativas siendo las siguientes las más relevantes:

- Davyw quantum (http://www.davyw.com/quantum), se trata de una aplicación web escrita en Javascript lo que la hace cómoda de utilizar desde cualquier lugar. Al estar programada en un lenguaje interpretado no es muy eficiente.
- Zeno (http://dsc.ufcg.edu.br/~iquanta/zeno/download_en.html), simulador escrito en Java muy semejante al que se presenta. Tan solo permite introducir una puerta por etapa y la aplicación colapsa si se utilizan demasiados qubits.
- jQuantum (http://jquantum.sourceforge.net/jQuantumApplet.html), otra alternativa escrita en Java que se puede utilizar directamente desde el navegador en forma de *applet*. Dispone de una útil visualización gráfica de los estados, la posibilidad de utilizar varios registros y es eficiente. Por el contrario la interfaz no es muy usable y no permite ver el progreso del estado.

Por el contrario, no se han encontrado alternativas para la simulación de un procesador *clásico-cuántico,* aunque se han desarrollado experimentos de este tipo a muy pequeña escala, como por ejemplo [5].

Sin embargo existen herramientas que permiten realizar computaciones *clásico-cuánticas*, algunas de alto nivel, lo que facilita en gran medida el desarrollo de algoritmos cuánticos:

- Q++ (http://sourceforge.net/projects/qplusplus/), extensión de C++ que permite ejecutar algunas instrucciones cuánticas simuladas. Se puede utilizar para probar rápidamente algoritmos cuánticos ya que al ser de alto nivel es fácil programar algoritmos complicados.
- QASM (http://www.media.mit.edu/quanta/qasm2circ/), lenguaje simple para programar circuitos cuánticos a bajo nivel, muy semejante a las instrucciones cuánticas que se han añadido al procesador del proyecto.

Existen otras muchas alternativas menos relevantes, una lista bastante exhaustiva de ellos se puede encontrar en [22].

Además, en [23] aparece un resumen de los modelos de simulación más exitosos, así como una breve introducción a la computación cuántica.

3 Desarrollo del proyecto

El proyecto completo está realmente divido en dos partes:

- El **simulador de circuitos cuánticos Qubit101**, que permite crear, editar y simular el comportamiento de esta representación de los algoritmos cuánticos utilizando un motor de simulación de estados cuánticos.
- El **simulador del procesador qMIPS**, que imita el comportamiento de un procesador MIPS I al que se le ha añadido una unidad funcional cuántica que utiliza las capacidades de encapsulación de Java para hacer uso del motor de simulación del programa Qubit101.

A continuación se describen tanto el motor de simulación como ambas herramientas tanto desde un punto de vista teórico como en su implementación.

3.1 Simulando los sistemas cuánticos

Este punto del proyecto es crítico, de ello depende la eficiencia del programa final en gran parte.

Realmente, no se conoce ningún algoritmo eficiente para simular los sistemas cuánticos (si se descubriera, invalidaría la ventaja de los computadores cuánticos), así que habrá que buscar una solución lo más eficiente posible a fin de minimizar el retraso en estos cálculos.

Este parte del proyecto está programada en el proyecto Java *Qubit101*, que es un simulador de circuitos cuánticos que utiliza la implementación que se explicará y de la que se aprovecha el proyecto *qMIPS*.

3.1.1 Representación matricial de los sistemas

Una primera idea surge rápidamente a raíz de la explicación expuesta sobre la computación cuántica. Los estados se pueden representar como vectores y los operadores como matrices, y es trivial realizar esta implementación en prácticamente cualquier lenguaje de programación:

- Se elige una representación para los estados, que serán vectores como hemos dicho. Por comodidad, lo más lógico es elegir la base computacional, que realiza simplemente:

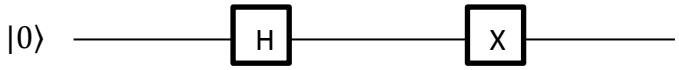
$$|0\rangle = \begin{pmatrix}1\\0\end{pmatrix}, |1\rangle = \begin{pmatrix}0\\1\end{pmatrix}$$

- Se busca una representación de las puertas cuánticas, que serán matrices como hemos dicho y se aplicaran multiplicándolas por el vector de estado correspondiente:

$$X = \begin{pmatrix}0 & 1\\1 & 0\end{pmatrix}, H = \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\1 & -1\end{pmatrix}, \dots$$

- Y se obtiene una representación para sistemas de muchos qubits. El producto tensorial en las matrices se conoce como *producto de Kronecker* y es relativamente fácil de implementar.

En esta representación la computación:



se calcularía:

$$XH|0\rangle = \begin{pmatrix}0 & 1\\1 & 0\end{pmatrix}\frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\1 & -1\end{pmatrix}\begin{pmatrix}1\\0\end{pmatrix} = \begin{pmatrix}0 & 1\\1 & 0\end{pmatrix}\frac{1}{\sqrt{2}}\begin{pmatrix}1\\1\end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\1\end{pmatrix}$$

Es en el último punto donde encontramos el problema de esta implementación: el producto de Kronecker hace crecer a las matrices y a los estados con el número de qubits en el sistema, para adecuarse a los nuevos grados de libertad. De hecho, para un número q de qubits, tendremos vectores de estado de 2^q elementos y matrices para los operadores de $2^q \times 2^q$ elementos, es decir, una complejidad exponencial en el número de qubits.

Esta primera implementación se realizó y fue descartada ya que con aproximadamente 10 qubits en el sistema, la máquina virtual de java era incapaz de reservar suficiente memoria para los estados y las matrices. Con 8 o 9 el tiempo de cálculo rondaba los minutos por operación.

3.1.2 Representación en mapa de componentes de los estados

El programa del proyecto utiliza una implementación radicalmente distinta. Esta implementación trata de alejarse del principal problema de la anterior: matrices de un tamaño desmesurado.

Si observamos los estados cuánticos en su forma matricial y su representación en notación de Dirac, por ejemplo:

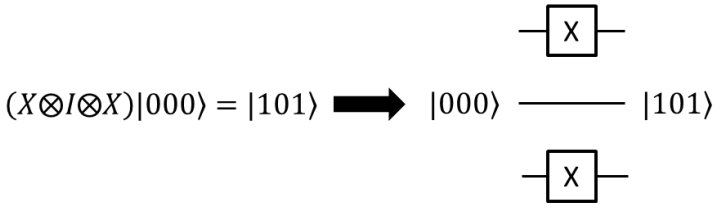
$$|101\rangle = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}$$

Podemos ver que parece mucho más cómodo trabajar con la notación de Dirac, donde solo utilizamos tres bits para representar lo mismo para lo que necesitamos ocho en la representación matricial.

Además, realizar operaciones sobre el vector requeriría una matriz con 64 datos para aplicarle, por ejemplo, una puerta X al tercer qubit:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

En la notación de Dirac tan solo necesitamos *interpretar* un símbolo que nos diga que puerta debemos aplicar:



Así, mientras en la representación matricial tenemos 64 datos para representar una puerta cuántica sobre tres qubits con un crecimiento exponencial en el número de qubits, en la representación de Dirac necesitamos tres símbolos y algún mecanismo que nos de la interpretación de ese símbolo que, en el caso de una implementación informática, será un trozo de código de tamaño constante, es decir, que no crece con el número de qubits. Hemos conseguido así pasar de un $O(2^n)$ en espacio a un orden constante para las puertas cuánticas.

La situación es más complicada si tenemos un estado en superposición. En la representación matricial, un estado de este tipo quedaría representado como un vector con dos componentes o más componentes no nulas:

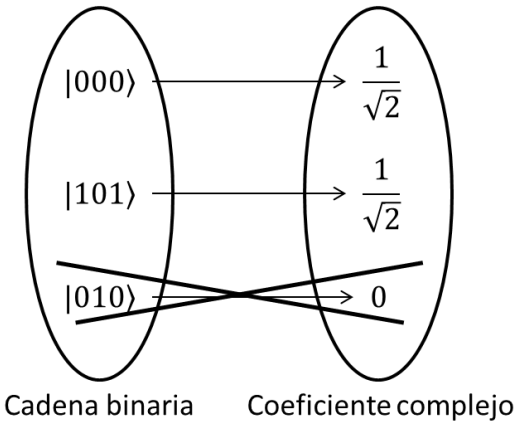
$$\begin{pmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ 0 \\ 0 \\ 0 \\ \frac{1}{\sqrt{2}} \\ 0 \\ 0 \end{pmatrix}$$

Equivalente en la notación de Dirac a:

$$\frac{1}{\sqrt{2}}|000\rangle + \frac{1}{\sqrt{2}}|101\rangle$$

Vemos que aquí la ventaja de esta notación se ha diluido ligeramente. La situación se agravará según aumentemos el *tamaño de la superposición* pero, al fin y al cabo, tendremos como límite máximo 2^q *datos* en total para la superposición máxima y en la notación matricial tendremos esa cantidad de *datos* sea como sea el tamaño de la superposición: hemos trasladado la complejidad del algoritmo a ese tamaño en vez del al número de qubits.

Pero, ¿Cómo son esos *datos* en cada representación? En la representación matricial estos datos serán una serie de coeficientes complejos en forma de vector. En la representación de Dirac tendremos una serie de parejas: una cadena binaria asociada a un coeficiente. Si este coeficiente es cero dicha pareja no aparece en la representación:



Así, tendremos qua almacenar estas parejas para tener toda la información del estado, de forma que un estado como:

$$\frac{1}{\sqrt{2}}|000\rangle + \frac{1}{\sqrt{2}}|101\rangle$$

Quedaría almacenado de la forma:

$$\left\{\left[000, \frac{1}{\sqrt{2}}\right], \left[101, \frac{1}{\sqrt{2}}\right]\right\}$$

En programación, la estructura de datos relevante es un *mapa*. Los mapas son estructuras de datos que asocian a cada *clave* un *valor*, de modo que si queremos obtener un valor concreto nos referimos a su clave asociada. Aquí, nuestras claves serán estas cadenas binarias y los valores asociados los coeficientes complejos.

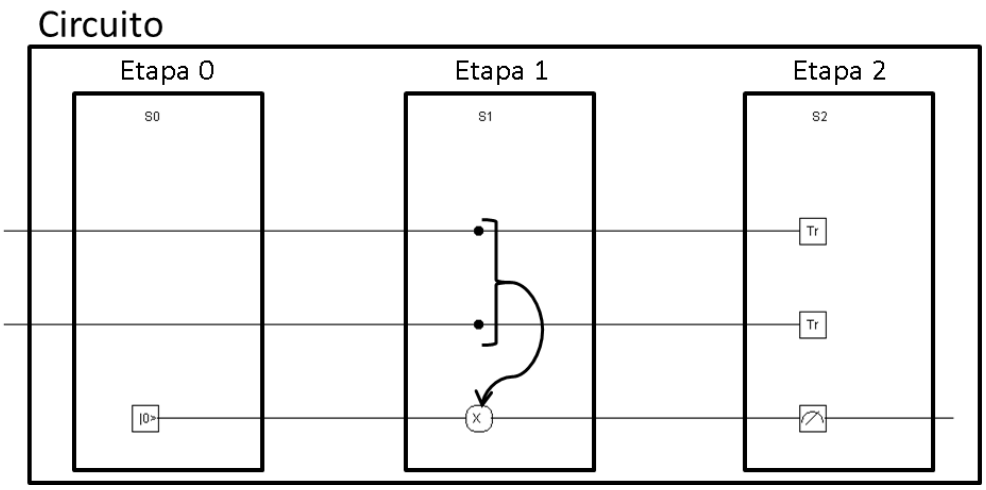
3.2 El simulador de circuitos cuánticos Qubit101

El punto de inicio del proyecto es el simulador de circuitos cuánticos Qubit101. Este simulador tiene implementado el motor de simulación de estados cuánticos que se presentó en la sección anterior.

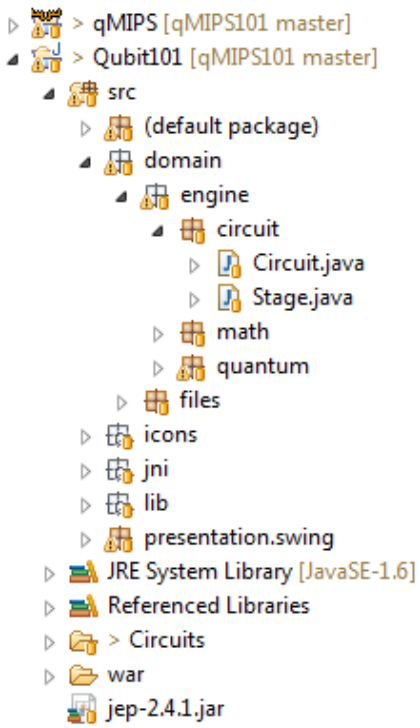
Este simulador es, básicamente, un editor gráfico de circuitos cuánticos que es capaz de generar, a partir de dicha representación gráfica, una sucesión de puertas cuánticas que simulen el efecto del circuito.

Un circuito es en la implementación un *array dinámico de etapas*, junto con la lógica de edición necesaria.

Estas etapas son, a su vez, colecciones de puertas cuánticas que se ejecutan, al menos conceptualmente, al mismo tiempo. Un qubit de control afectará a todas las puertas controladas de su misma etapa:

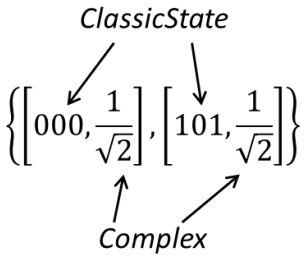


Estas estructuras están implementadas en las clases *Stage* para las etapas y *Circuit* para los circuitos:



Las etapas además, tienen implementada la lógica de simulación en sus métodos *Simulate*, luego para ejecutar un circuito completo tan solo tenemos que ir ejecutando etapa a etapa pasando la salida de una a la entrada de la siguiente.

Esta simulación se realiza utilizando el esquema de mapa descrito en la sección anterior. Concretamente, las cadenas binarias *clave* son de la clase *ClassicState* y los coeficientes binarios *valor* son instancias de la clase *Complex*:

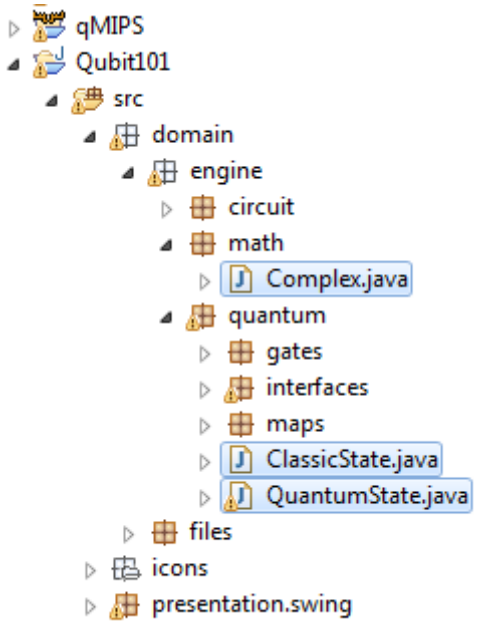


$$\left\{\left[000, \frac{1}{\sqrt{2}}\right], \left[101, \frac{1}{\sqrt{2}}\right]\right\}$$

La eficiencia de la simulación dependerá en gran medida de la implementación de los mapas que se utilice. Java proporciona en su API una implementación de mapas ordenados en forma de un tipo muy eficiente de árbol binario que se conoce como *árbol rojo-negro*, esta implementación nos asegura un tiempo logarítmico en el número de elementos para obtenerlos o insertarlos. La clase de la API de Java que representa estos árboles se llama TreeMap.

En la clase *QuantumState*, está implementada la lógica de los estados cuánticos. Al añadir un componente a la superposición el algoritmo busca si esta componente ya está en el estado y suma los coeficientes. Si no estuviera simplemente añade la componente al mapa. En todo caso, lo elimina si el coeficiente resulta ser cero.

Así están ubicadas estas tres clases ya descritas en la estructura de paquetes del proyecto:



Queda ver como se aplican las puertas cuánticas sobre esta representación de estados. Como hemos dicho anteriormente, en notación de Dirac representamos las puertas con símbolos y tan solo necesitamos interpretarlos para aplicarlas. En este caso, solo necesitamos decirle al simulador como actúa la puerta sobre cada componente de la superposición, de forma que crear el estado resultante consista en recorrer todo el estado de entrada generando las nuevas componentes. De esta forma, el mismo trozo de código que representa una puerta X para un solo qubit, funciona igualmente para cientos de ellos.

Entrando en la implementación, todas las puertas *unitarias* actúan de una forma muy parecida y tienen una plantilla que pueden extender. Esta plantilla encapsula la lógica para recorrer las componentes o utilizar qubits de control y el programador tan solo debe definir cómo actúa la puerta sobre cada componente.

La plantilla es la clase *UnitaryGateTemplate*. Una vez extendida solo habría que implementar el método abstracto *singleComponentOperation,* que, por ejemplo, para la puerta X sería:

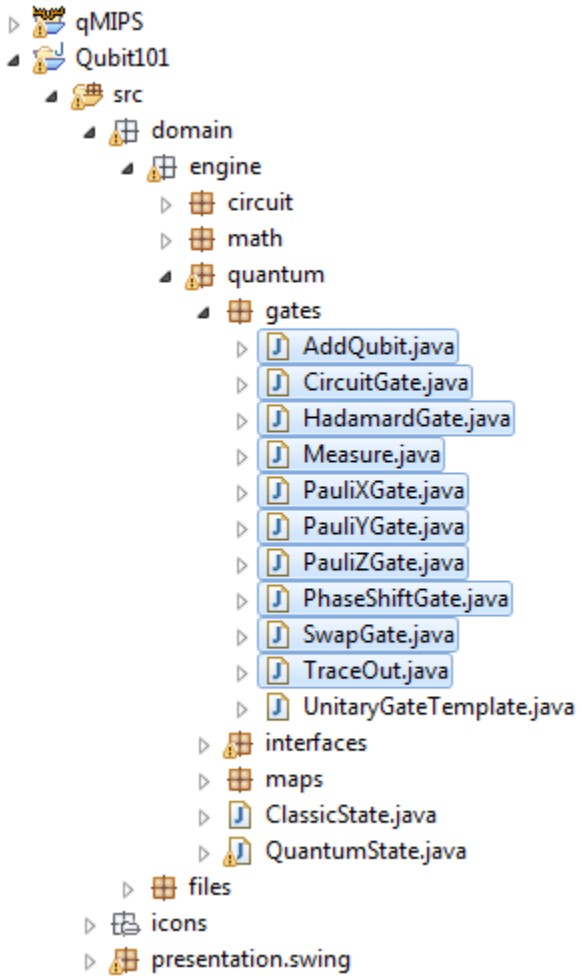
```
public void singleComponentOperation(Entry<ClassicState, Complex> e, int targetQubit, QuantumState res)
{
    byte state[] = e.getKey().getState();
    byte stateneg[] = (byte[])state.clone();
    stateneg[targetQubit] = (byte)(stateneg[targetQubit] != 0 ? 0 : 1);
    res.add(e.getValue(), new ClassicState(stateneg));
}
```

Las puertas que no son unitarias no pueden implementar esta plantilla ya que no pueden ser operaciones controladas. Por suerte, no se utilizan muchas de ellas.

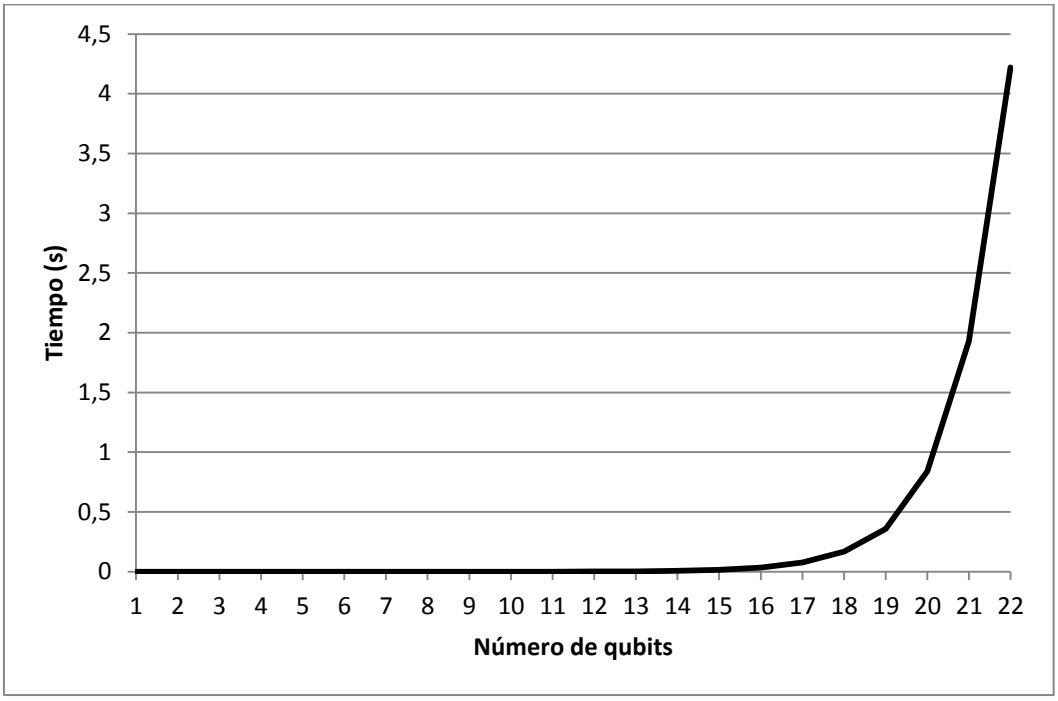
El simulador tiene suficientes puertas cuánticas para ser una máquina cuántica universal y de hecho es suficiente con menos, pero sería incomodo trabajar con un conjunto de puertas muy reducido. Estas puertas ya implementadas son las siguientes:

- *AddQubit:* Se encarga de añadir un qubit más al circuito, en la posición que se le indique. Esta puerta no es unitaria.
- *CircuitGate:* Esta es una puerta especial, encapsula todo un circuito cuántico en su interior a modo de subrutina.
- *HadamardGate:* La puerta de Hadamard.
- *Measure:* Se encarga de realizar las medidas sobre los estados. Dispone de una variable consultable con el resultado de la medida. Esta puerta no es unitaria.
- *PauliXGate:* La puerta X, correspondiente a un inversor clásico.
- *PauliYGate:* La puerta correspondiente a la matriz Y de Pauli.
- *PauliZGate:* La puerta correspondiente a la matriz Z de Pauli.
- *PhaseShiftGate:* Puerta de cambio de fase, rota la fase del estado dependiendo de un parámetro de entrada.
- *SwapGate:* Intercambia el estado de dos qubits del sistema.
- *TraceOut:* Descarta un qubit del sistema. Previamente lo mide para romper cualquier entrelazamiento entre el qubit afectado y los demás. Esta puerta no es unitaria.

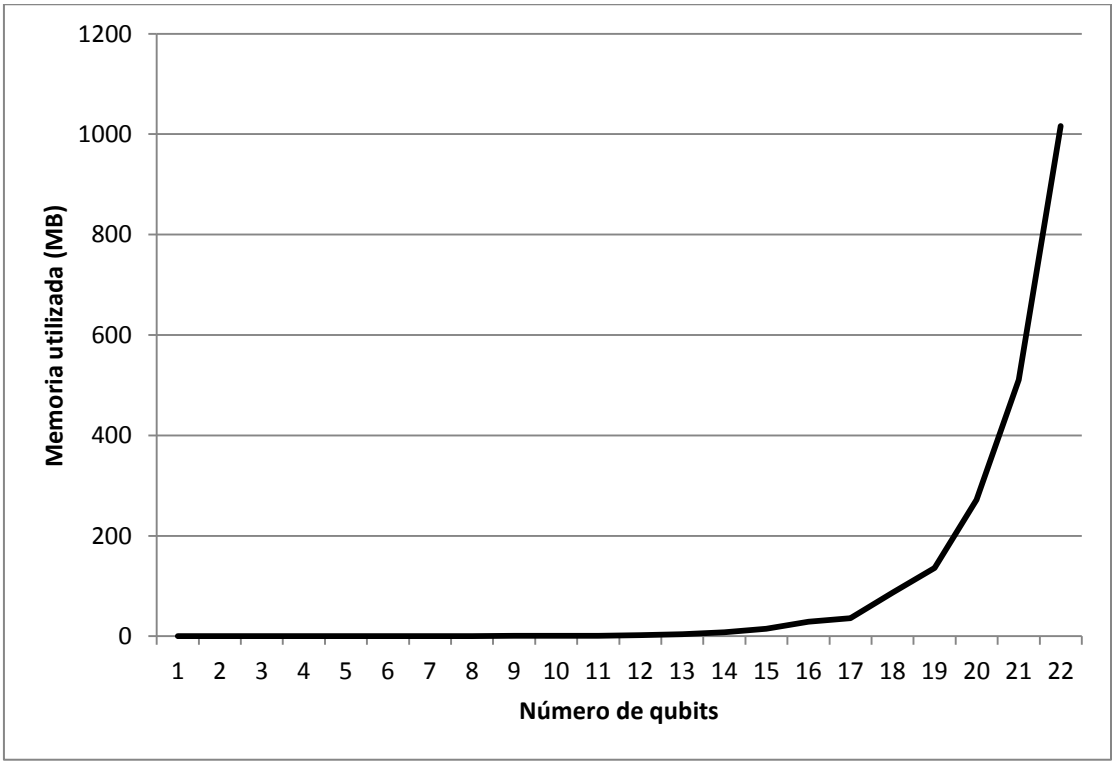
En la estructura de paquetes del simulador estas puertas están almacenadas en las clases de la imagen:



En esta representación se han conseguido simular hasta 22 qubits en superposición antes de que la máquina virtual de Java se quede sin memoria. El sistema tarda unos 4 segundos en crear dicha superposición. Las gráficas siguientes se han obtenido aplicando puertas de Hadamard a cada vez más qubits, de forma que se va desarrollando una superposición cada vez de mayor tamaño:



Se puede observar aquí claramente el comportamiento exponencial de la complejidad del algoritmo. En cuanto a la memoria que ocupa el estado el comportamiento es igualmente exponencial, y la máquina virtual de Java lanza una excepción cuando se alcanza un tamaño por defecto de aproximadamente un GB:



Este motor de simulación del simulador Qubit101 será importado y utilizado por el proyecto qMIPS para simular la unidad funcional cuántica.

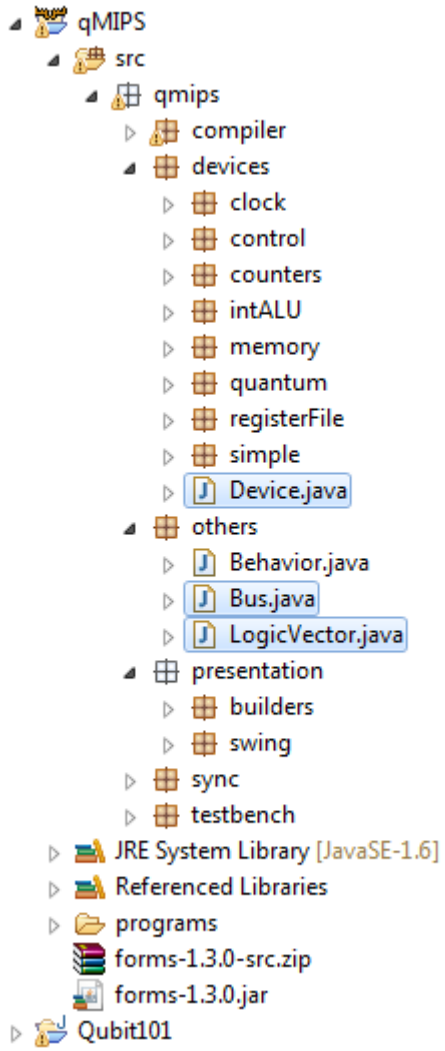
3.3 Simulando hardware en Java

El sistema debe ser capaz de imitar el hardware de la maquina sin poner ninguna restricción con respecto a qué tipo de componente de está simulando. Para ello se ha construido un marco en el que resulta sencillo construir estos componentes, usando un diseño parecido a lo que aportan los lenguajes específicamente para descripción de hardware como VHDL o Verilog.

3.3.1 Definiendo los dispositivos

En el sistema, todo componente debe heredar de la clase *Device*, una clase abstracta que encapsula toda la lógica común a todos los componentes. El programador solo debe encargarse de definir cómo reacciona el componente a los cambios en sus entradas.

Los componentes se interconectan con buses, que pueden tener uno o varios cables. Son instancias de la clase *Bus*. En los bus se escriben y se leen vectores lógicos; arrays de datos booleanos que indican el nivel de tensión del cada cable. Estos arrays son instancias de la clase *LogicVector.*



Cada componente debe tener como parámetros de su constructor los buses, tanto de entrada como de salida, de los que quiera disponer y se interconectan compartiendo los buses al instanciarlos.

Para definir el comportamiento de los buses, se debe implementar dentro de los hijos de *Device* la función *defineBehavior()* que debe ser llamada al final del constructor además. Dentro de esta función tendremos una o varias llamadas a la función *behavior(…)*, donde se indicará a los cambios de que buses debe reaccionar el componente y qué hacer ante esos cambios.

Mostraré como ejemplo cómo construir un registro síncrono:

- Se declara la clase heredada de Device y se colocan los buses como atributos, para poder utilizarlos para definir el comportamiento:

```
public class SynchronousRegister extends Device {

    Bus input, output, en, clk, rst;
```

- Se define el constructor, con todos los buses como parámetros de entrada, se asignan a sus atributos y se llama al método *defineBehavior():*

```
public SynchronousRegister(Bus input, Bus output, Bus en, Bus rst, Bus clk) {
    this.input = input;
    this.output = output;
    this.en = en;
    this.rst = rst;
    this.clk = clk;
    disp =  new SynchronousRegisterDisplay();
    defineBehavior();
}
```

- Ahora definimos el comportamiento del dispositivo, implementando la función *defineBehavior().* En su interior encontramos dos llamadas a *behavior(..).* La primera, indica qué ocurre al darse un ciclo de subida en el reloj. Para eso colocamos como primer parámetro un array con tan solo el bus correspondiente al reloj (*clk*), ya que solo debe reaccionar así ante cambios en este bus. Como segundo parámetro se introduce el comportamiento en si, como una clase hija de *Behavior* de la que debemos implementar el método *task()*. El método comprueba que realmente se está en un ciclo de subida leyendo el reloj, además comprueba que la señal de habilitación está activada. Si se dan ambas, escribe y mantiene en su salida lo que tenga a la entrada.

```
behavior(new Bus[] { clk }, new Behavior() {

    @Override
    public void task() {
         if (clk.read().get(0)) {
            if (en.read().get(0)) {
                disp.setContent(input.read());
                output.write(input.read());
            }
        }
    }

});
```

- La segunda llamada a *behavior(…)* se usa para reiniciar el registro si se activa la señal de reset.

```
behavior(new Bus[]{ rst }, new Behavior() {

    @Override
    public void task() {
        if (rst.read().get(0)) {
            output.write(new LogicVector(output.size()));
            disp.setContent(new LogicVector(output.size()));
        }
    }

});
```

Cabe señalar que el reloj que se usa en el sistema no hereda de la clase *Device* ya que se trata de un dispositivo especial, que no debe reaccionar a ningún evento de escritura sino que debe disparar dichos eventos de forma espontánea.

3.3.2 Sincronizando los dispositivos en ejecución

Cada dispositivo de un sistema construido con este marco se ejecuta en su propio hilo. Al registrarse las tareas que debe realizar un dispositivo, se asocian a los buses correspondientes dichas tareas, de forma que si algo escribe en uno de ellos esas tareas se realizan y provocarán otras escrituras que, a su vez, pondrán en marcha más.

Cuando un sistema de un gran tamaño está en ejecución, puede tener un gran número de tareas corriendo al mismo tiempo y que deben ejecutarse en un orden estricto. El proyecto está construido de forma que es fácil cambiar la forma en la que se sincronizan los dispositivos, se dispone de una o varias clases de sincronización que implementan la interfaz *Synchronization*. Qué implementación se utiliza se define en la clase *SyncShortcut* y en el proyecto se utiliza la clase *PoolSync.*

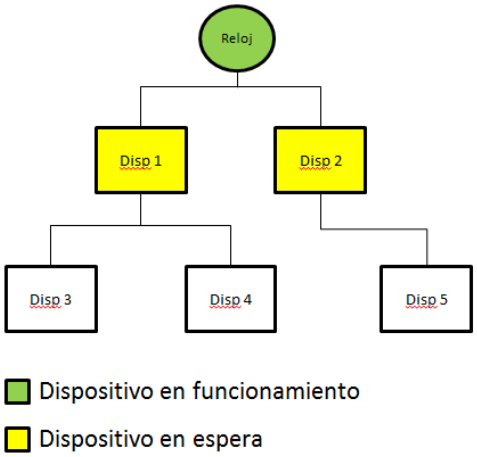
En dicha clase se utiliza lo que se denomina una *piscina de hilos* para administrar los hilos en ejecución. Una piscina de hilos es una herramienta a la que se envían una serie de tareas y se encarga de ejecutarlas en una serie de hilos que se crean tan sólo una vez y si no tienen nada que hacer se duermen. Existen piscinas con un número de hilos fijo, donde las tareas esperan a tener un hilo disponible para ejecutarse y piscinas que crecen con el número de tareas. En el proyecto se usa esta segunda implementación, de forma que si hay más tareas que hilos se crean nuevos hilos y si ocurre al contrario habrá hilos durmiendo a la espera de trabajo. Si pasa una cantidad fijada de tiempo y algún hilo no ha trabajado se destruye.

Utilizar una piscina de hilos:

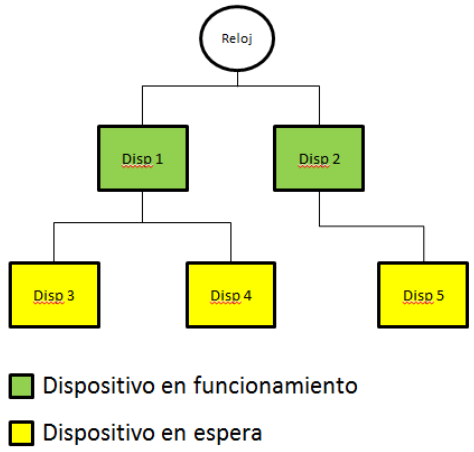
a) Evita la carga de trabajo extra que significa crear los hilos. Tan solo se crean en la primera iteración del programa y en las sucesivas están ya creados y disponibles.
b) Simplifica el manejo de los hilos. El programador ya no tiene que preocuparse de crear los hilos y sincronizarlos ya que está todo encapsulado en la piscina.

En esta implementación los ciclos de reloj tienen una duración muy variable, desde microsegundos para operaciones aritméticas hasta minutos u horas para operaciones de tipo cuántico, por tanto no es válido que el reloj emita flancos de forma periódica en el tiempo. El funcionamiento de la clase *PoolSync* es el siguiente:

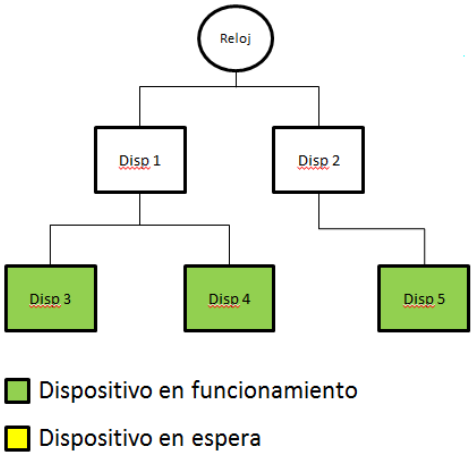
1. El reloj emite un flanco por alguna interacción externa, que normalmente será la interfaz de usuario, activando una serie de tareas de los dispositivos que reaccionan al reloj, los síncronos.



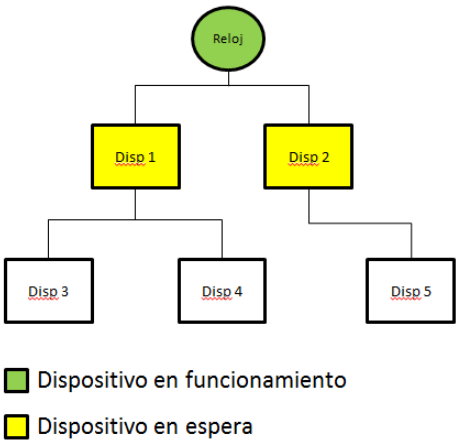
2. Las tareas recién despertadas no se ejecutan de inmediato sino que esperan en una cola a que todos los dispositivos anteriores hayan terminado.



3. Cada vez que una tarea concluya informa a la clase de sincronización que lleva una cuenta del número de ellas que están en funcionamiento.



4. Cuando no quedan más tareas en funcionamiento se activan las que estaban en espera que a su vez pondrán en la cola más tareas.
5. En el momento en el que no queden más tareas en funcionamiento ni en la cola de espera, se considera que el sistema ha llegado a un estado estable y se despierta al reloj para que emita otro flanco.



Utilizando este sistema se garantiza que las tareas se ejecutarán en el orden correcto y se podrá esperar si alguna lleva un tiempo muy largo. Además el número de hilos en la piscina tras el primer ciclo será tan grande como el número máximo de tareas que se lancen a la vez y en general no tendrá que crecer.

3.4 Diseñando el procesador qMIPS

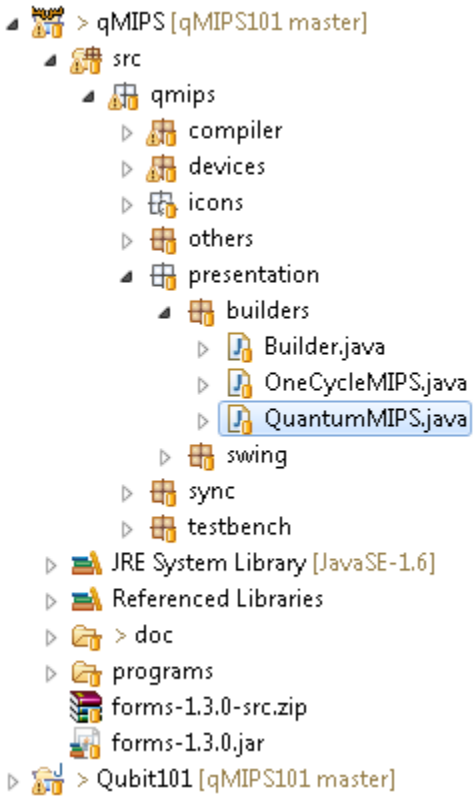
El procesador diseñado en el proyecto está basado en una versión simplificada del MIPS clásico, sin encadenamiento de instrucciones, que utiliza una cantidad variable de ciclos por instrucción, presentado de forma educativa en el libro [2]. Al presentado en el libro se han añadido varias instrucciones, siendo bastante cercano al procesador MIPS I.

El procesador tiene las siguientes características:

- Bus de instrucciones y bus de datos de 32 bits.
- Fichero de registros de 32 registros de 32 bits. El registro 0 no contiene ningún valor, siempre vale cero.
- Memoria común de datos e instrucciones. No se ha implementado jerarquía de memoria y se supone que responde en un solo ciclo.
- Todas las operaciones son sobre 32 bits. No se aceptan instrucciones de *byte* o *halfword*.
- No dispone de unidad de punto flotante, solo usa aritmética entera.

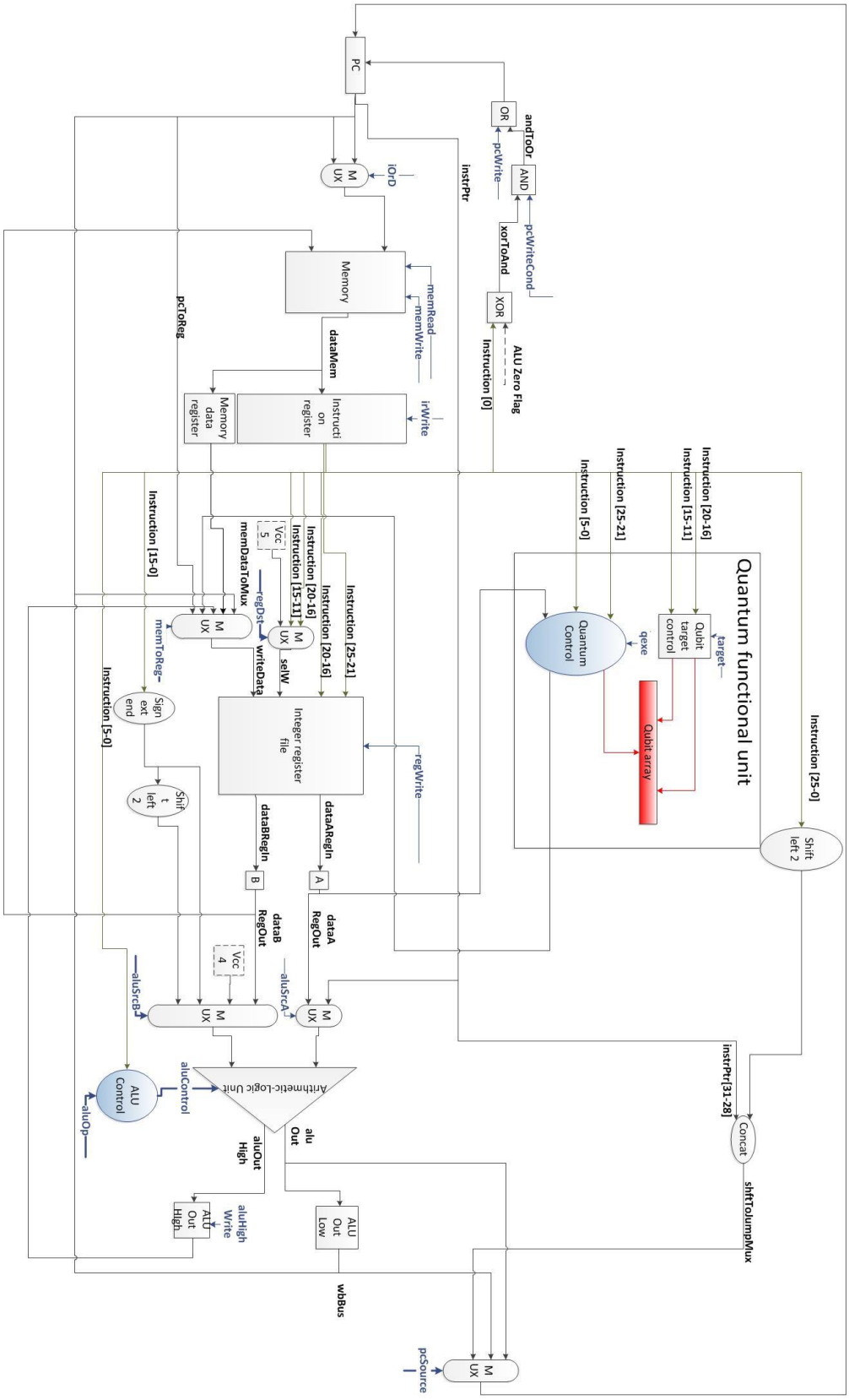
3.4.1 Arquitectura "física" del sistema

Uno de los objetivos del proyecto es presentar una arquitectura real de un procesador, implementable físicamente, no simplemente un intérprete de instrucciones. Esta arquitectura está programada en la clase *QuantumMIPS*:



En esa clase se interconectan todos los dispositivos que conforman el procesador y se asignan todas las variables que necesita la interfaz gráfica para mostrar la información relevante al usuario.

Esta interconexión se entiende más fácilmente en el siguiente esquema:

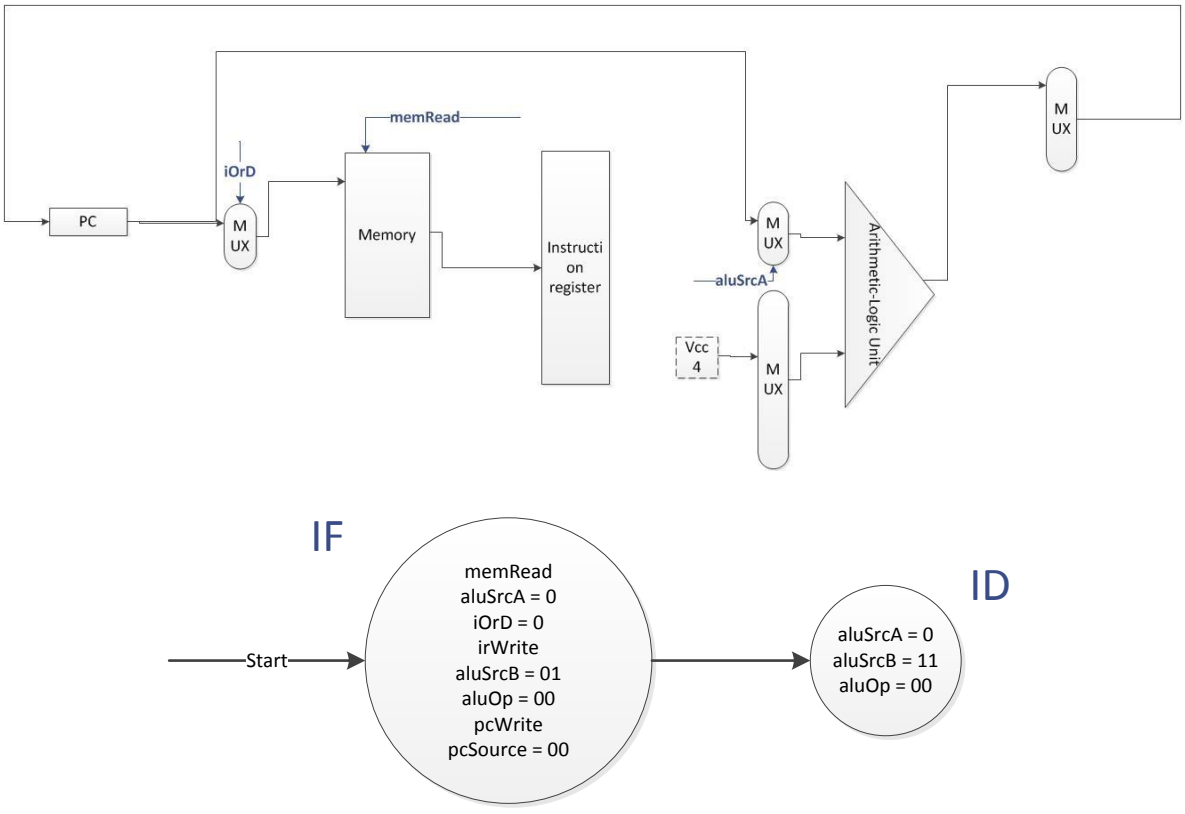

PC
OR
andToOr
AND
pcWrite
pcWriteCond
xorToAnd
XOR
ALU Zero Flag
Instruction [0]
instrPtr
M UX
iOrD
Memory
memRead
memWrite
dataMem
pcToReg
Memory data register
Instruction register
irWrite
Instruction [5-0]
Instruction [25-21]
Instruction [20-16]
Instruction [15-11]
Instruction [15-0]
Quantum functional unit
Quantum Control
qexe
Qubit target control
target
Qubit array
Shift left 2
Instruction [25-0]
Vcc 5
regDst
memDataToMux
memToReg
writeData
selW
Sign ext end
Shift left 2
Integer register file
regWrite
dataARegIn
dataBRegIn
A
B
dataA RegOut
dataB RegOut
Vcc 4
aluSrcA
aluSrcB
Arithmetic-Logic Unit
ALU Control
aluControl
aluOp
alu Out
aluOut High
ALU Out High
aluHigh Write
ALU Out Low
wbBus
instrPtr[31-28]
Concat
shftToJumpMux
pcSource

En esta figura las líneas verdes representan a la instrucción de memoria, las azules son señales de control, las rojas son operaciones cuánticas y el resto son negras.

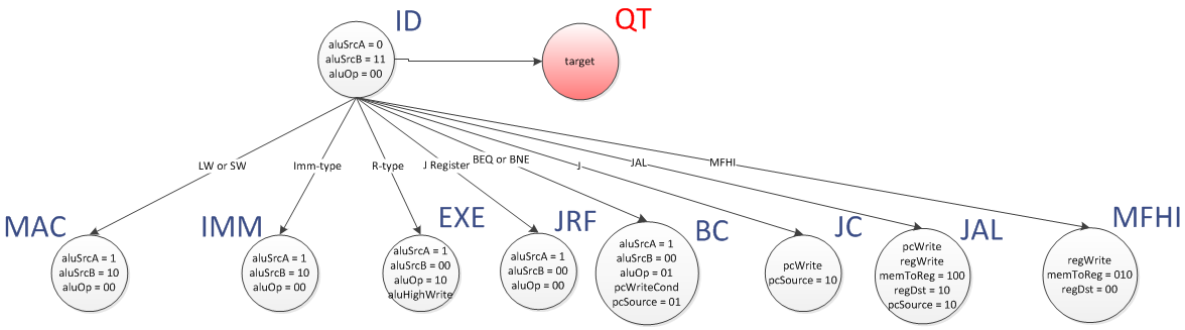
3.4.2 Fases de la unidad de control

El procesador qMIPS, a diferencia del MIPS I real, no utiliza encadenamiento de instrucciones, pero tampoco una ejecución de tipo monociclo. Las instrucciones se segmentan tal y como si existiera encadenamiento pero ya que tenemos todo el hardware disponible para cada instrucción, podemos adecuar su longitud en ciclos a lo estrictamente necesario. Por tanto, la unidad de control está programada para seguir un árbol de estados provocando que cada instrucción realice tan solo las operaciones que necesita.

El primer ciclo de ejecución se denomina fase IF (Instruction Fetch). Consiste siempre en traer la instrucción siguiente de la memoria y calcular el puntero de instrucción a la siguiente asumiendo que no hay ningún salto.

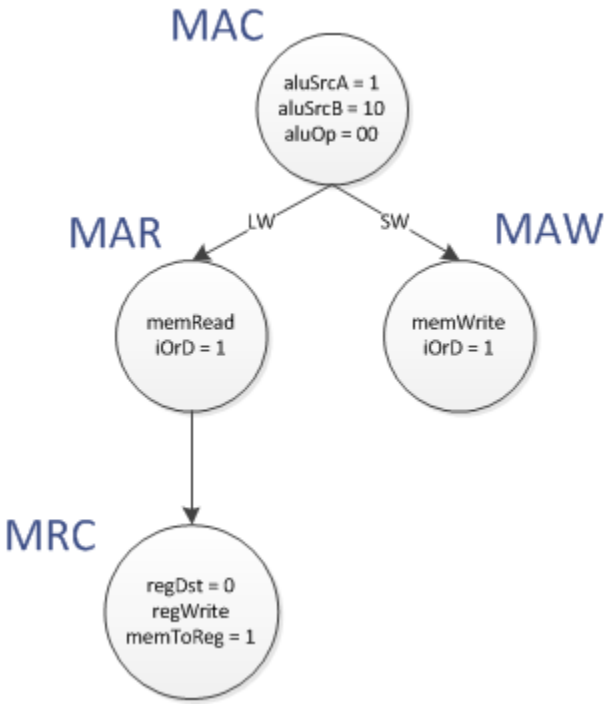


El segundo ciclo de ejecución se denomina fase ID (Instruction Decode). En esta fase la unidad de control debe decidir a partir del código de operación qué señales debe activar para poner en funcionamiento la instrucción que corresponda. Además, se precalcula la dirección de un posible salto.

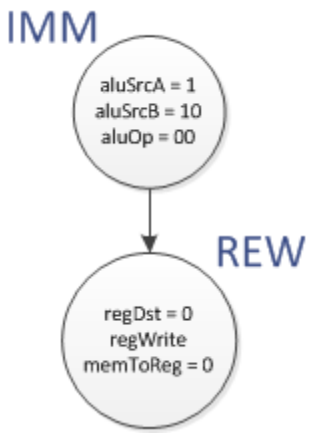


A partir de este momento la fase siguiente es muy distinta dependiendo del tipo de instrucción.

- Operaciones con memoria: La primera fase de este tipo de operaciones consiste en calcular la dirección efectiva donde se debe operar (Fase MAC). Para ello se indica a la ALU que sume el contenido del registro indicado con el dato inmediato que aparece en la instrucción. A continuación, según sea la instrucción de lectura (Fase MAR) o de escritura (Fase MAW) se activa la señal correspondiente de la memoria. Por último, si se trata de una operación de lectura, debemos escribir el resultado de la operación en el registro indicado (Fase MRC).

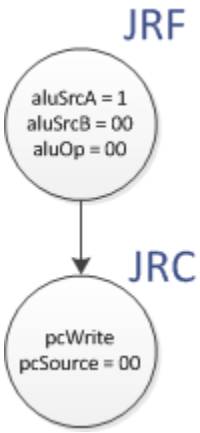


- Suma con operando inmediato: En su primera fase (Fase IMM) se ordena a la ALU que sume el dato inmediato de 16 bits de memoria con su signo extendido con el contenido del registro indicado. En la segunda fase simplemente se escribe el resultado en el registro correspondiente (Fase REW).



- Operación lógico-aritmética básica: Este tipo de instrucción es muy similar al anterior. Se ordena a la ALU que realice la operación indicada en los últimos 5 bits de la instrucción sobre los registros indicados (Fase EXE) y a continuación se escribe el dato en el fichero de registros (Fase RC).

- Saltos a dirección apuntada por registro: En la primera fase (Fase JRF) se ordena a la ALU que deje pasar el valor del registro indicado hacia el contador de programa. En el siguiente ciclo simplemente se escribe en dicho contador el valor correspondiente (Fase JRC).



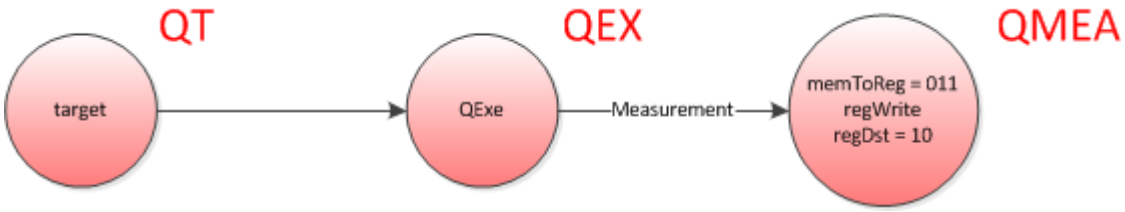
- Salto condicional: Dado que la dirección efectiva del salto se calculó en la fase ID, en este ciclo (Fase BC) tan solo debemos comprobar utilizando la ALU si la condición del salto se cumple y escribir el contador de programa en tal caso.

- Salto incondicional: Igualmente que en el caso anterior la dirección efectiva del salto ya está calculada, así que en este caso tan solo debemos escribir sobre el contador de programa (Fase JC).

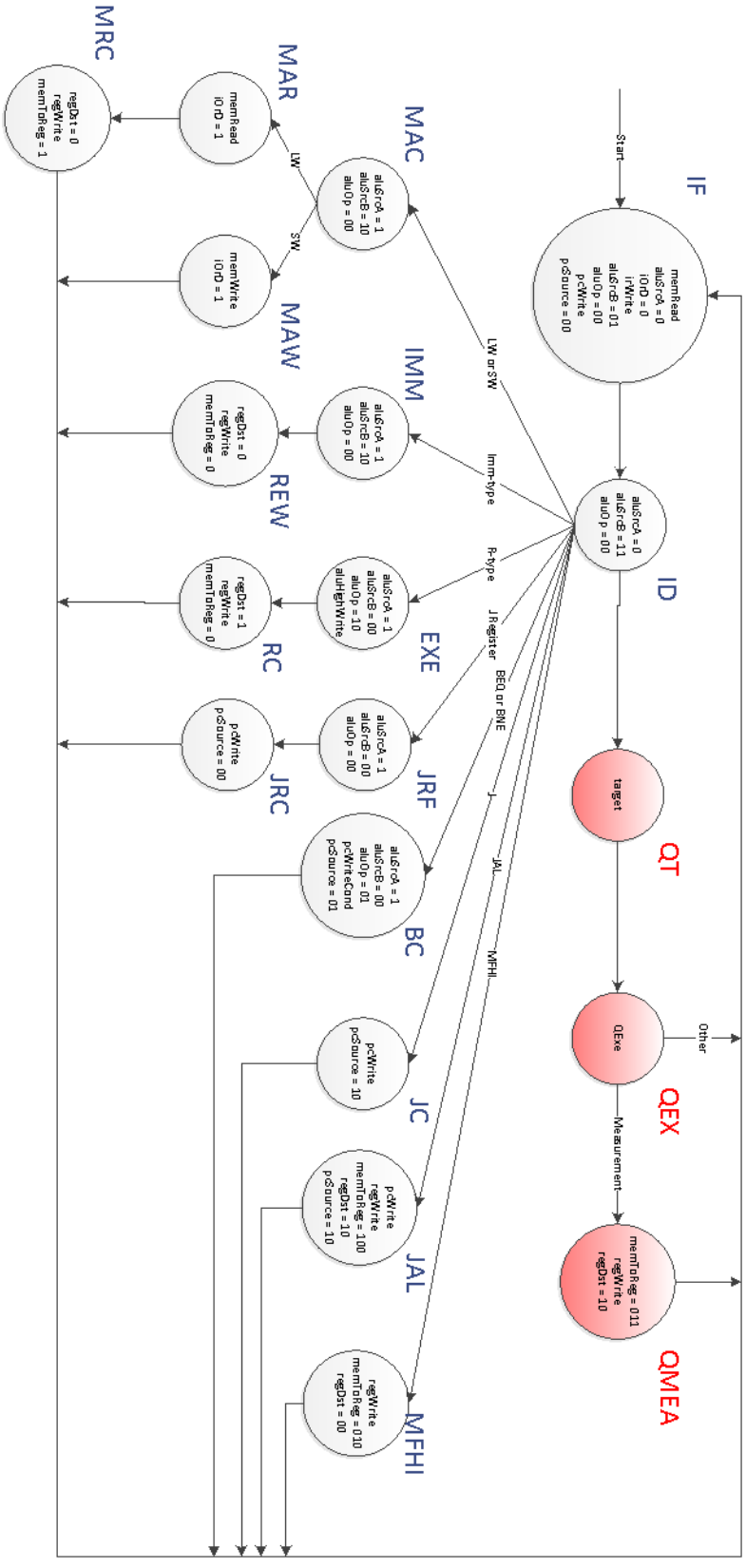
- Salto a subrutina: Idéntico al salto incondicional salvo que en este caso debemos indicarle al fichero de registros que escriba el puntero de instrucción actual en el registro 31, para poder realizar el retorno de subrutina (Fase JAL).

- Mover desde el registro de parte alta: En esta fase (MFHI) tan solo debemos mover el dato contenido en el registro de parte alta de la ALU hacia el registro indicado en la instrucción.

- Por último quedarían las operaciones de tipo cuántico. Estas fases se describen en detalle en la sección siguiente.



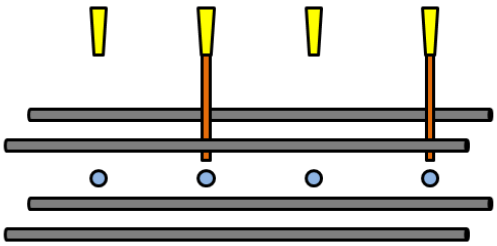
El árbol de estado completo de la unidad de control quedaría, por tanto, de la siguiente forma:


Start
IF
memRead
aluSrcA = 0
iOrD = 0
irWrite
aluSrcB = 01
aluOp = 00
pcWrite
pcSource = 00
ID
aluSrcA = 0
aluSrcB = 11
aluOp = 00
LW or SW
Imm-type
R-type
JRegister
BEQ or BNE
J
JAL
MFHI
MAC
aluSrcA = 1
aluSrcB = 10
aluOp = 00
IMM
aluSrcA = 1
aluSrcB = 10
aluOp = 00
EXE
aluSrcA = 1
aluSrcB = 00
aluOp = 10
aluRegWrite
JRF
aluSrcA = 1
aluSrcB = 00
aluOp = 00
BC
aluSrcA = 1
aluSrcB = 00
aluOp = 01
pcWriteCond
pcSource = 01
JC
pcWrite
pcSource = 10
JAL
pcWrite
regWrite
memToReg = 100
regDst = 10
pcSource = 10
MFHI
regWrite
memToReg = 010
regDst = 00
MAR
memRead
iOrD = 1
MAW
memWrite
iOrD = 1
REW
regDst = 0
regWrite
memToReg = 0
RC
regDst = 1
regWrite
memToReg = 0
JRC
pcWrite
pcSource = 00
MRC
regDst = 0
regWrite
memToReg = 1
LW
SW
target
QT
QExe
QEX
Other
Measurement
memToReg = 011
regWrite
regDst = 10
QMEA

3.4.3 La unidad funcional cuántica

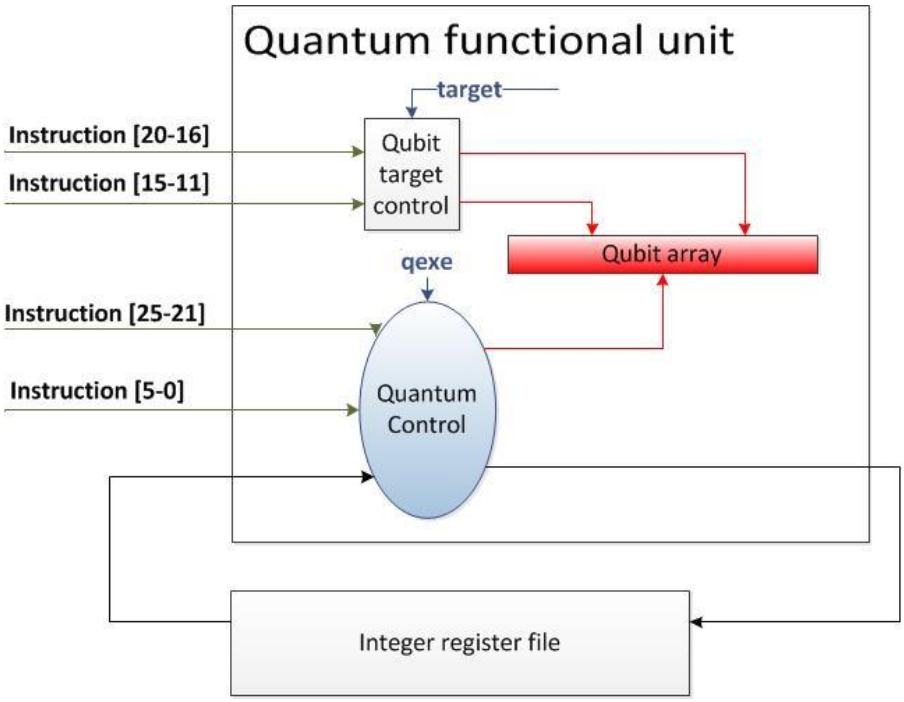
Este punto es el punto fundamental en el desarrollo del proyecto. Dado que no podemos hacer ninguna asunción sobre el funcionamiento de la unidad cuántica, no podemos fijarnos a ninguna implementación física concreta. Aun así, para que sea más fácil visualizar su comportamiento, vamos a asumir que se trata de una trampa de iones.

En las trampas de iones se dispone de una serie de átomos atrapados por campos electromagnéticos oscilantes, que confinan cada partícula espacialmente. Una serie de haces láser se encargan de enviar señales electromagnéticas adecuadas para operar con cada ion de la trampa.



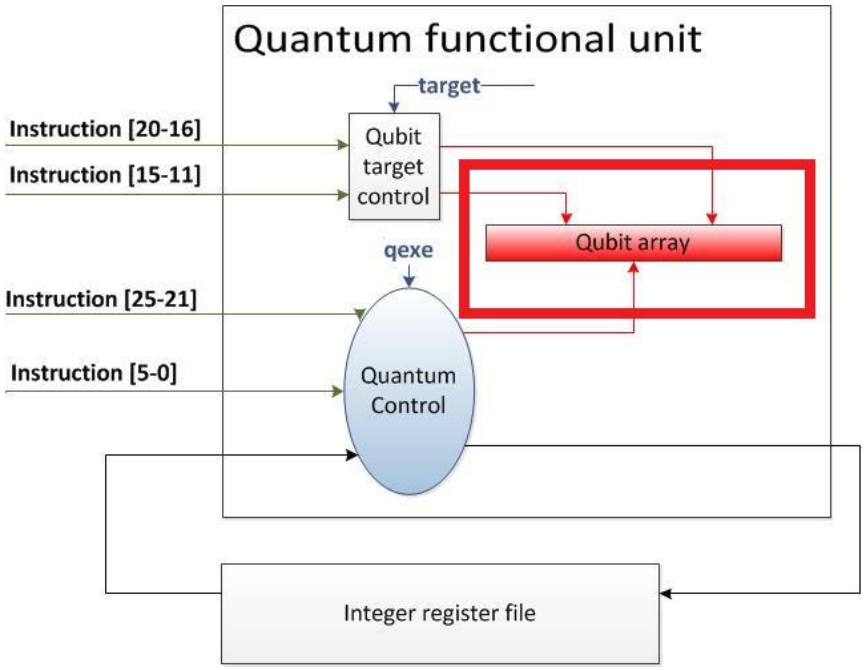
La imagen superior representaría una trampa de iones de 4 qubits.

La unidad funcional cuántica implementada en el proyecto tiene la siguiente estructura:

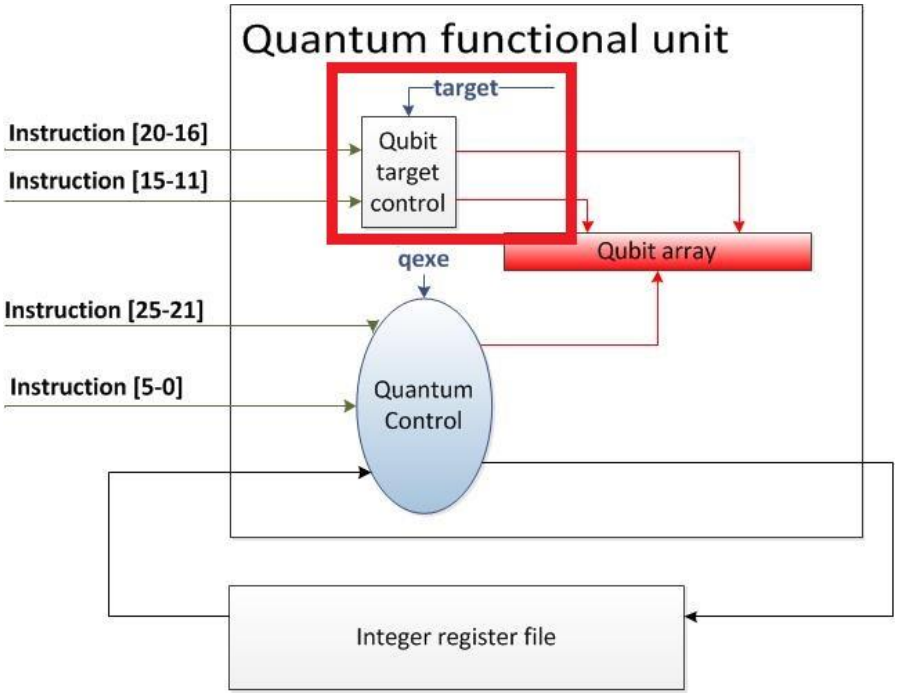


Consta de tres partes principales:

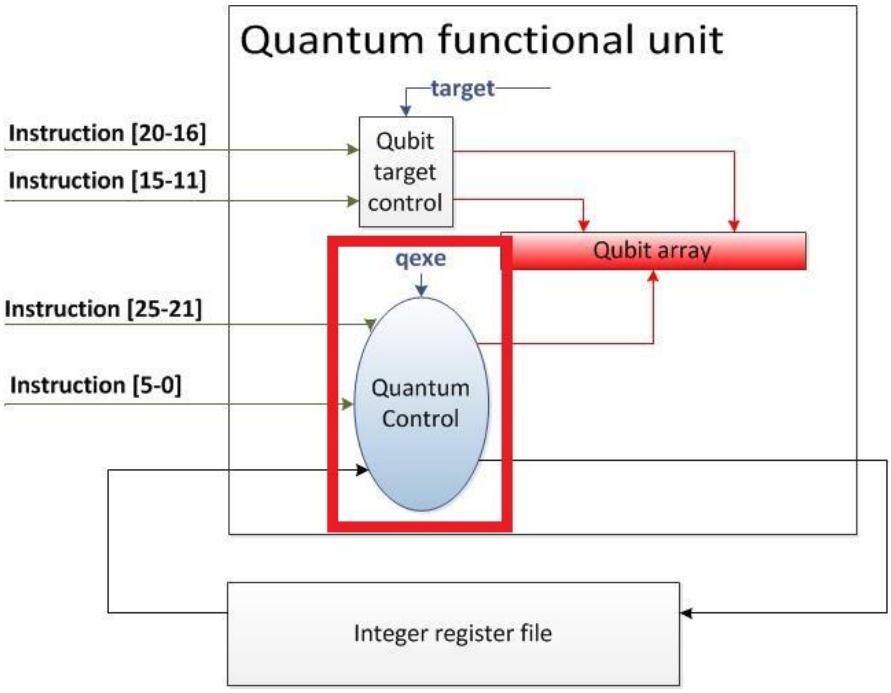
- Un *array de qubits* que en el caso del simulador contiene 32 qubits, cada uno de ellos se etiqueta con una 'Q' seguida de un número, de forma que para referirnos al primer qubit escribiremos Q0, Q1 para el segundo, etc. Hasta llegar al Q31. Cada uno de estos qubits sería un ión de la trampa.



- Un sistema de apuntado, que se encarga de desplazar el láser sobre el qubit correspondiente. Este sistema obtiene información sobre a qué qubit debe apuntar de la instrucción. Se encarga de señalar el qubit objetivo y, si no son ambos el mismo, el qubit de control. Debe quedar claro que esta imagen de los láseres es ficticia, se trata simplemente de *preparar* los qubits sobre los que se va a actuar.

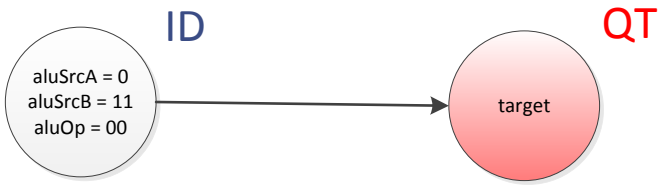


- Una unidad de control cuántica, que dependiendo de la instrucción actual decida qué *onda* deben enviar los láseres de forma que tenga el efecto de la puerta cuántica requerido en el qubit apuntado.

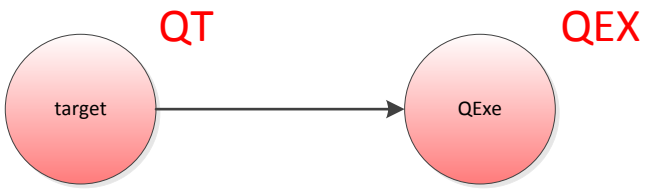


El proceso a la hora de ejecutar una instrucción de tipo cuántico sería el siguiente:

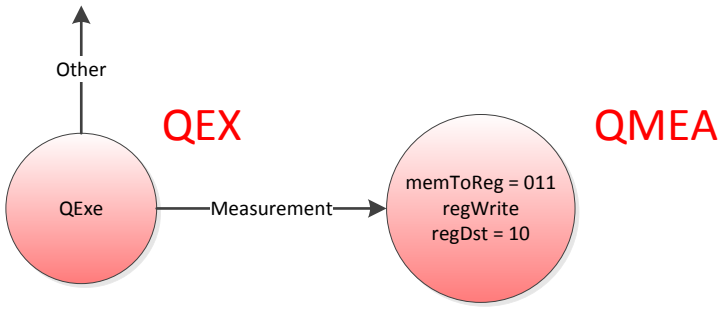
1. La instrucción se trae de memoria (IF) y se decodifica (ID) como cualquier otro tipo de instrucción.
2. La unidad de control detecta la instrucción de tipo cuántico y ordena a la unidad de apuntado que debe preparar los qubits seleccionados (**fase QT**). Estos qubits vienen identificados en la propia instrucción: los cables del 20 al 16 (5 cables), indican qué qubit es el objetivo. Los cables del 15 al 11 (otros 5) indican cuál es el de control. Si ambos son iguales la unidad de apuntado interpreta que no hay qubit de control.



3. En el siguiente ciclo la unidad de control ordena a la unidad de control cuántica que realice las operaciones seleccionadas (**fase QEX**). Toma la operación seleccionada de los cables de la instrucción del 0 al 5. Además, esta unidad necesita los cables 21 al 25 por si se utiliza algún parámetro que forma parte de la instrucción así como acceso al fichero de registros por si el parámetro procede de este. De esta unidad parte el único bus de salida de la unidad cuántica que escribirá en el registro correspondiente los resultados de una posible medida.



4. Si la instrucción era una operación cuántica corriente, se procede a ejecutar la siguiente instrucción. Si se trata de una medida, se inicia la fase de medida (**fase QMEA**). En esta fase se ordena al fichero de registros que escriba el resultado de la medida.



5. Se ejecuta la instrucción siguiente.

3.4.4 Detalles sobre el "array de qubits"

En este simulador el array de qubits que se utiliza tiene un ancho de 32 qubits, sin embargo, en la sección 4.1.2 se obtuvo que el motor de simulación es incapaz de sostener más de 22 qubits en superposición sin consumir toda la memoria de la máquina virtual de Java , esto sin tener en cuenta que con ese tamaño de superposición cada operación dura unos 4 segundos, lo que puede hacer bajar el límite aún más si queremos que la computación se lleve a cabo en un tiempo razonable. Aun así el tamaño del array es mucho mayor, lo que podría llevar a la aplicación a colapsarse si no se tiene el debido cuidado.

Estos qubits de más están disponibles para que el programador no tenga problemas de espacio en cuanto a qubits auxiliares. En todo caso se debe tener cuidado de mantener baja la superposición de estados todo lo que sea posible, por ejemplo, midiendo qubits que no vayan a ser necesarios más adelante en la computación. Cada vez que realicemos una optimización de este tipo estaremos reduciendo a la mitad el tiempo de computación.

Además, la tendencia en computación clásica es que las operaciones se ejecuten sobre datos con un *tamaño de palabra* prefijado, es decir, que las unidades funcionales realicen los cálculos sobre pares de datos del tamaño de dicha palabra, que suele coincidir con el tamaño de los registros y de las instrucciones.

En el procesador MIPS I, por ejemplo, el tamaño de palabra es de 32 bits y las operaciones se realizan sobre registros de ese tamaño. Si queremos realizar una suma de tan solo un bit con otro debemos utilizar dos registros de 32 bits cada uno.

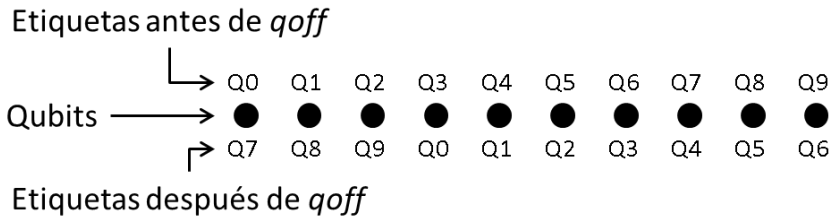
Aquí se está operando sobre qubits independientes, es como si en un computador clásico se ejecutaran las instrucciones aritméticas típicas bit a bit: sumar el bit 2 con el 3, restar el 4 del 1, etc. ¿Por qué se no se utilizan lo que se suele denominar en información cuántica *registros cuánticos*? Por tres razones:

- Como se ha señalado antes, la complejidad de la simulación no nos permite crear superposiciones de más de 22 qubits sin que la máquina virtual de Java deje de funcionar. Esta simulación almacena aun así, 32 qubits. Se podrían utilizar, por ejemplo, 4 registros de 8 qubits pero parece obvio que son demasiado pocos registros para realizar operaciones de una cierta complejidad. En el caso contrario nos encontraríamos con muchos registros pero demasiado pequeños para que sean verdaderamente útiles.
- La mayoría de los algoritmos cuánticos están muy bien definidos en forma de circuitos cuánticos y estos se suelen referir a qubits independientes en muchos

momentos. Si utilizáramos qubits agrupados, tendríamos que utilizar todo un registro para usar un solo qubit y con un número tan pequeño de ellos no sería práctico.

- También con respecto a los circuitos cuánticos, referirnos a cada uno de ellos independientemente nos da una transformación de circuito a código máquina prácticamente directa.

Aun así, una instrucción de la unidad funcional cuántica permite imitar fácilmente el comportamiento de *registros cuánticos* de un tamaño variable. La instrucción *qoff* permite desplazar el etiquetado de los qubits a la posición que se desee, por ejemplo al ejecutar *qoff R1*, si R1 contiene un 3:



Así, podemos guardar los índices de inicio de cada registro y programar subrutinas genéricas desplazando el origen del etiquetado a dichos índices.

3.4.5 Las instrucciones y el compilador

El procesador presentado en [2] aceptaba tan solo un subconjunto de las instrucciones que se han implementado. Aun así este no es un procesador MIPS clásico propiamente dicho ya que algunas instrucciones funcionan de una forma ligeramente distinta.

El compilador se ha desarrollado con la herramienta ANTLR [13]. A esta herramienta se le proporcionan gramáticas en un lenguaje específico y construye "parseadores" en Java automáticamente.

Es bastante complejo construir gramáticas de una cierta complejidad con esta herramienta y, al no ser este el objetivo del trabajo, el compilador realiza muy pocas comprobaciones más allá de que la gramática del programa sea la correcta y es bastante rígido. Si el programa está mal escrito o contiene alguna incoherencia como por ejemplo una dirección negativa o fuera de rango, el programa fallará.

A continuación se muestran las instrucciones clásicas que acepta el compilador; Rd, Rs y Rt son registros, C representa un número entero con signo de 16 bits:

Instrucción	C. Op.	Func.	Resumen
Instrucciones Aritmético-Lógicas			
add Rd, Rs, Rt	0x0	0x20	Suma de los registros Rs y Rt en Rd. Lanza una excepción si hay desbordamiento.
addu Rd, Rs, Rt	0x0	0x21	Suma de los registros Rs y Rt en Rd. Ignora el desbordamiento.
sub Rd, Rs, Rt	0x0	0x22	Resta de los registros Rs y Rt en Rd. Lanza una excepción si hay desbordamiento.
subu Rd, Rs, Rt	0x0	0x23	Resta de los registros Rs y Rt en Rd. Ignora el desbordamiento.
mult Rd, Rs, Rt	0x0	0x18	Multiplica Rs y Rt en Rd. Si Rs y Rt contienen más de 16 bits, el resultado puede ocupar más de los 32 bits que caben en Rd y la parte alta se almacenara en el registro *High*.
div Rd, Rs, Rt	0x0	0x1A	Realiza la división entera de Rs entre Rt en Rd. Almacena el resto en el registro *High*.
divu Rd, Rs, Rt	0x0	0x1B	Realiza la división entera de Rs entre Rt en Rd. Almacena el resto en el registro *High*.
and Rd, Rs, Rt	0x0	0x24	Realiza la operación logica Y entre Rs y Rt en Rd.
or Rd, Rs, Rt	0x0	0x25	Realiza la operación logica O entre Rs y Rt en Rd.
xor Rd, Rs, Rt	0x0	0x26	Realiza la operación logica XOR entre Rs y Rt

			en Rd.
nor Rd, Rs, Rt	0x0	0x27	Realiza la operación logica O negada entre Rs y Rt en Rd.
slt Rd, Rs, Rt	0x0	0x2A	Coloca un 1 en Rd si Rs es menor a Rt.
Instrucciones con operando inmediato			
addi Rd, Rs, C	0x8	-	Suma Rs y C en Rd. Lanza una excepción si hay desbordamiento.
Operaciones con memoria			
lw Rd, C(Rs)	0x23	-	Carga en Rd el contenido de la memoria en la dirección Rs + C.
sw C(Rd), Rs	0x2B	-	Escribe en la dirección Rd + C de la memoria el contenido de Rs.
Instrucciones de salto			
jr Rs	0x1B	-	Salta a la dirección almacenada en el registro Rs.
j C (o etiqueta)	0x02	-	Salta a la dirección especificada en C.
jal C (o etiqueta)	0x03	-	Salta a la dirección especificada en C y almacena en R31 la dirección de la siguiente instrucción. Se utiliza para realizar saltos a subrutina. Para realizar el retorno de subrutina se ejecuta jr R31.
beq Rs, Rt, C (o etiqueta)	0x04	-	Si Rs y Rt son iguales, avanza o retrocede el número de instrucciones especificados en C.
bne Rs, Rt, C (o etiqueta)	0x05	-	Si Rs y Rt son distintos, avanza o retrocede el número de instrucciones especificadas en C.
Excepciones			
trap C	0x1A	-	Lanza la excepción de código C.
Especiales			
mfhi Rs	0x1C	-	Mueve el valor del registro *High* al registro Rs.

Las operaciones cuánticas que acepta el compilador son las siguientes. Qt y Qc son qubit objetivo y de control respectivamente, si son el mismo la operación es no controlada:

Instrucción	C. Op.	Func.	Resumen
Operaciones cuánticas unitarias			
qhad Qt, Qc	0x0C	0x00	Puerta de Hadamard.
qx Qt, Qc	0x0C	0x01	Puerta X de Pauli. Inversor.
qy Qt, Qc	0x0C	0x02	Puerta Y de Pauli.
qz Qt, Qc	0x0C	0x03	Puerta Z de Pauli.
qphs Qt, Qc, Rs	0x0C	0x10	Cambio de fase especificado en el registro Rs.
qnph Qt, Qc, Rs	0x0C	0x11	Cambio de fase negativo especificado en el registro Rs.

Operaciones cuánticas no unitarias			
qmea Qt, Rs, S	0x0F	0x1A	Mide el qubit Qt y lo vuelca en Rs desplazado a la izquierda el valor de S, de 5 bits.
qrst Rs	0x0C	0x1B	Limpia el estado cuántico y vuelca en él contenido de Rs.

Aunque con las instrucciones anteriores la máquina cuántica ya es universal, las siguientes dos instrucciones hacen mucho más sencillo la ejecución de ciertos algoritmos, sobre todo las subrutinas cuánticas:

Instrucción	C. Op.	Func.	Resumen
Operaciones cuánticas de control			
qoff Rs	0x0C	0x1D	Utiliza como qubit 0 el qubit señalado por los últimos 5 bits del registro, es decir, si Rs contiene un 7, Q0 pasará a ser el que anteriormente era el Q7; el Q1 pasara al Q8, etc. Para deshacer los cambios simplemente hay que hacer qoff R0.
qcnt Rs	0x0C	0x1C	Marca el qubit apuntado por los últimos 5 bits del registro Rs como qubit de control, por ejemplo, si el el registro Rs contiene un 5, el qubit Q5 controlara el resto de operaciones cuánticas unitarias que sigan a esa, a no ser que el qubit objetivo sea también Q5. Se pueden marcar tantos qubits de control al mismo tiempo como se quiera. Para desmarcar uno de ellos, se repite la instrucción sobre el mismo qubit.

El compilador, además, acepta dos directivas:

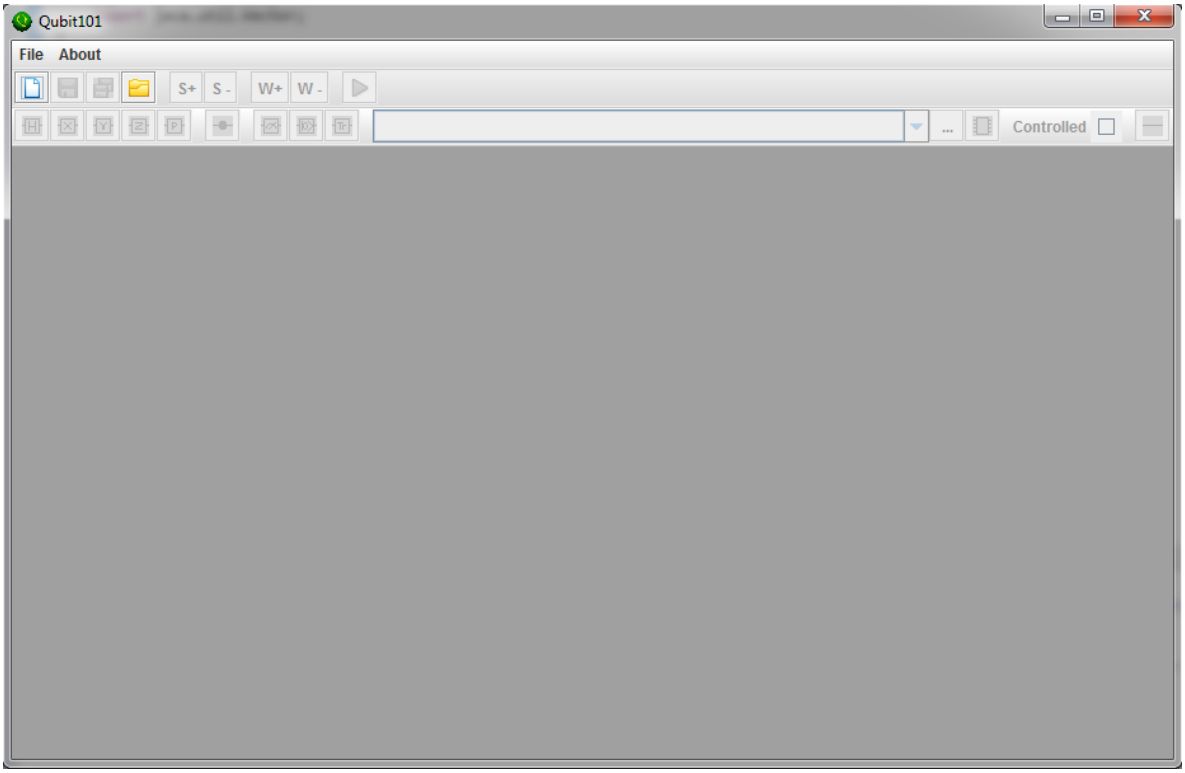
- *.word* A B: Almacena en la dirección de memoria A el dato B de 32 bits. Se utiliza para guardar datos en memoria antes de iniciar la ejecución.
- *.text* A: Tras esta directiva se deben colocar las instrucciones del programa. La primera instrucción estará en la dirección A de memoria.

4 Manual de uso

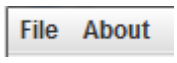
El proyecto se compone de dos partes, el simulador de circuitos cuánticos Qubit101 y el simulador del procesador MIPS con funciones cuánticas qMIPS. A continuación se describen ambas herramientas a modo de manual de uso.

4.1 Simulador de circuitos cuánticos Qubit101

La pantalla principal del simulador siguiente:



Se compone de un menú superior:



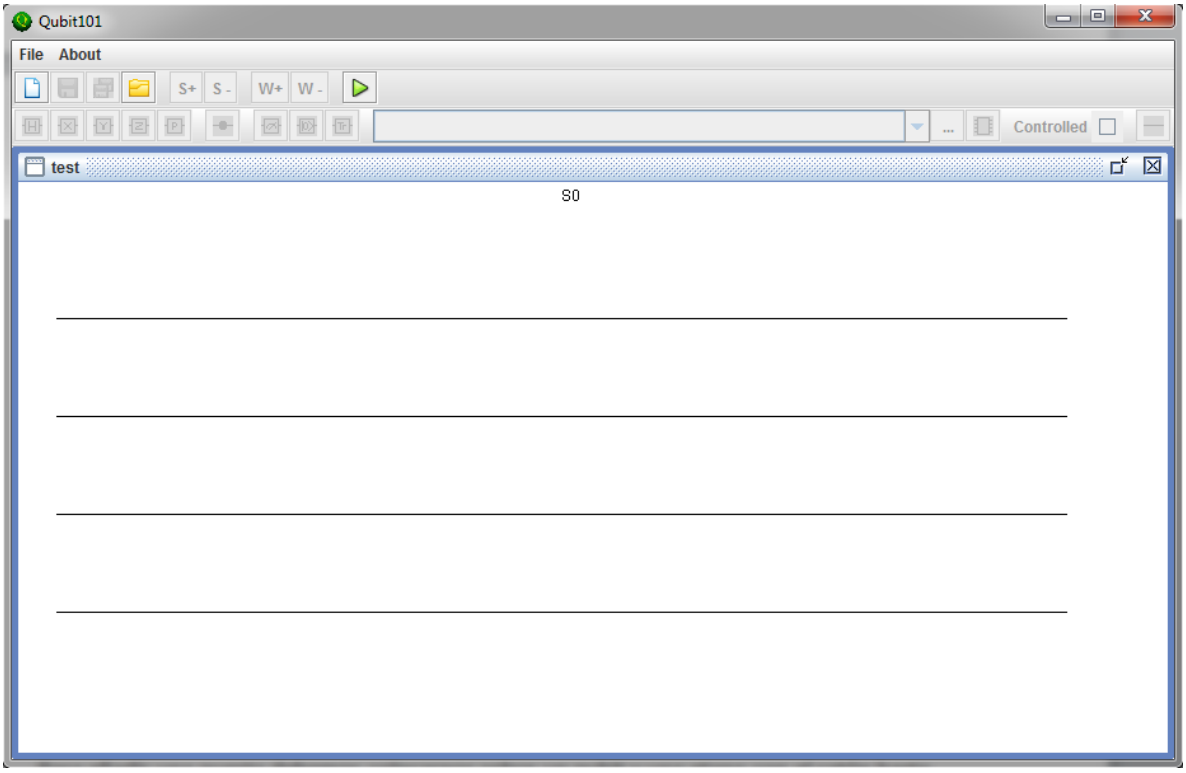
Una barra de herramientas:

Y una zona de trabajo que ocupa el resto de la pantalla.

4.1.1 Diseñando un nuevo circuito

Para empezar a diseñar un nuevo circuito, se debe pulsar el botón  o el elemento del menú *Nuevo…*

Esto abrirá una ventana que nos pedirá que introduzcamos un nombre para el circuito así como el número de qubits de los que dispondrá (se pueden añadir o quitar posteriormente). Al aceptar se abrirá en el espacio de trabajo una nueva ventana con el circuito creado:



Los circuitos se organizan por etapas. En cada etapa se puede añadir una puerta cuántica por qubit y las etapas se pueden añadir o eliminar utilizando los botones **S+** y **S-**.

Para añadir una puerta debemos colocarnos sobre un qubit y una etapa con el ratón hasta que se muestre un pequeño recuadro punteado y hacer click, esto resaltara el qubit en la etapa seleccionada y nos permitirá añadir puertas cuánticas con los botones de la barra de herramientas.

Las puertas cuánticas unitarias simples de las que se dispone para formar circuitos son las siguientes:

-  : Puerta de Hadamard

- : Puerta X de Pauli.
- : Puerta Y de Pauli.
- : Puerta Z de Pauli.
- : Puerta de cambio de fase. Para esta puerta se debe especificar un parámetro α para la fase.

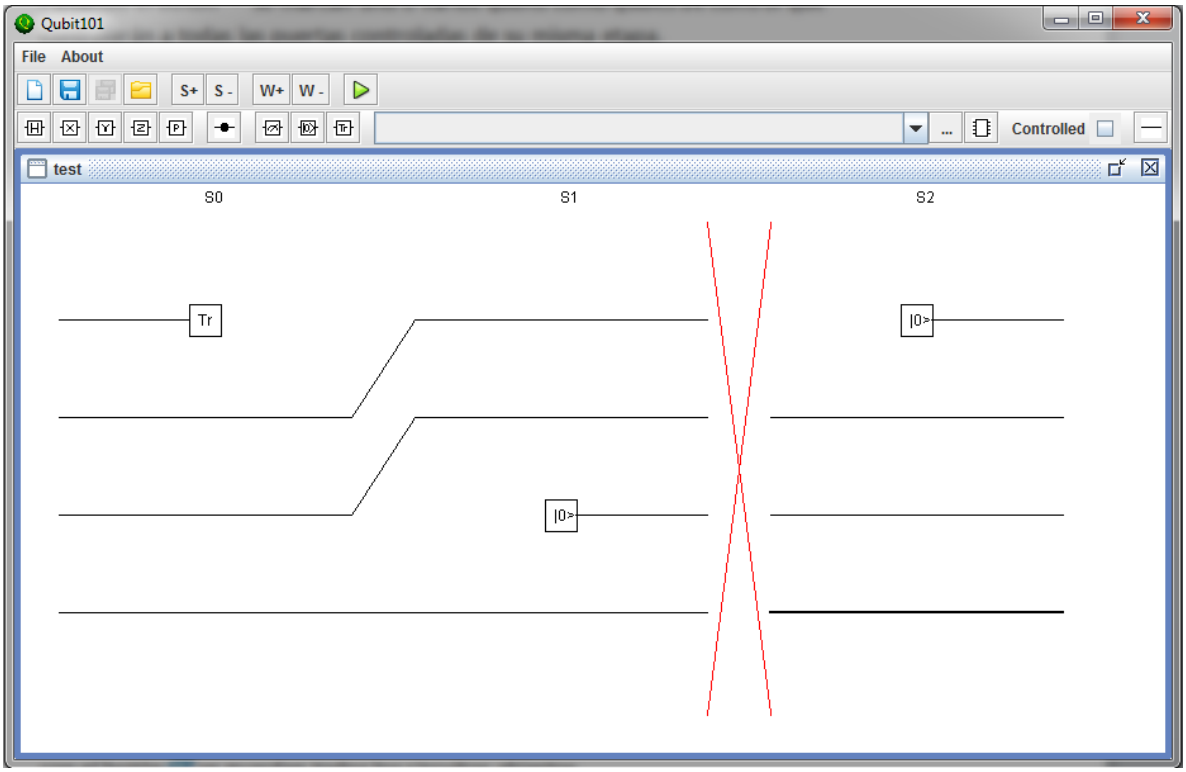
Todas las puertas anteriores se pueden hacer controladas marcando la casilla Controlled antes de añadirlas al circuito. Para identificarlas como controladas se mostraran de forma circular en vez de cuadradas.

Pulsando el botón se marcan uno o varios qubits como qubits de control que controlarán a todas las puertas controladas de su misma etapa.

Se dispone de tres puertas no unitarias:

- : Realiza una medida en el qubit y la etapa seleccionadas.
- : Añade un qubit auxiliar a la etapa seleccionada.
- : Mide y descarta el qubit seleccionado en esa etapa.

Con estas últimas dos puertas se debe cuidar que al finalizar una etapa e iniciarse la siguiente el número de qubits debe coincidir. Si no coinciden el programa lo señalara con un aspa roja entre las etapas conflictivas:



Se pueden añadir o quitar qubits de cada etapa utilizando los botones **W+** y **W-**.

Por último, para eliminar una puerta cuántica basta seleccionarla y pulsar el botón —.

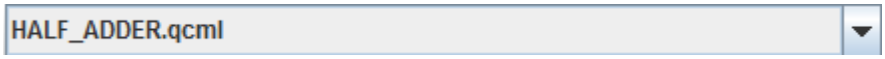
4.1.2 Guardando y cargando circuitos

El programa permite guardar los circuitos en formato QCML (*Quantum Circuit Markup Language,* extensión de XML). Pulsando el botón  se guarda el circuito seleccionado y con el botón  se guardan todos los circuitos abiertos.

Para cargar un circuito se pulsa el botón  y al seleccionarlo se abrirá una nueva ventana con el circuito guardado.

4.1.3 Utilizando circuitos cuánticos como puertas

El simulador permite utilizar un circuito guardado como si de una puerta cuántica más se tratase. Primero se debe cargar el circuito utilizando el botón … y el circuito seleccionado aparecerá en el menú desplegable:



Si se selecciona el circuito en dicho menú y se pulsa el botón  se añadirá el circuito como puerta.

Dado que el circuito que se utilice puede tener más de un qubit, se debe cuidar que el circuito seleccionado quepa en el actual y no sobresalga por debajo.

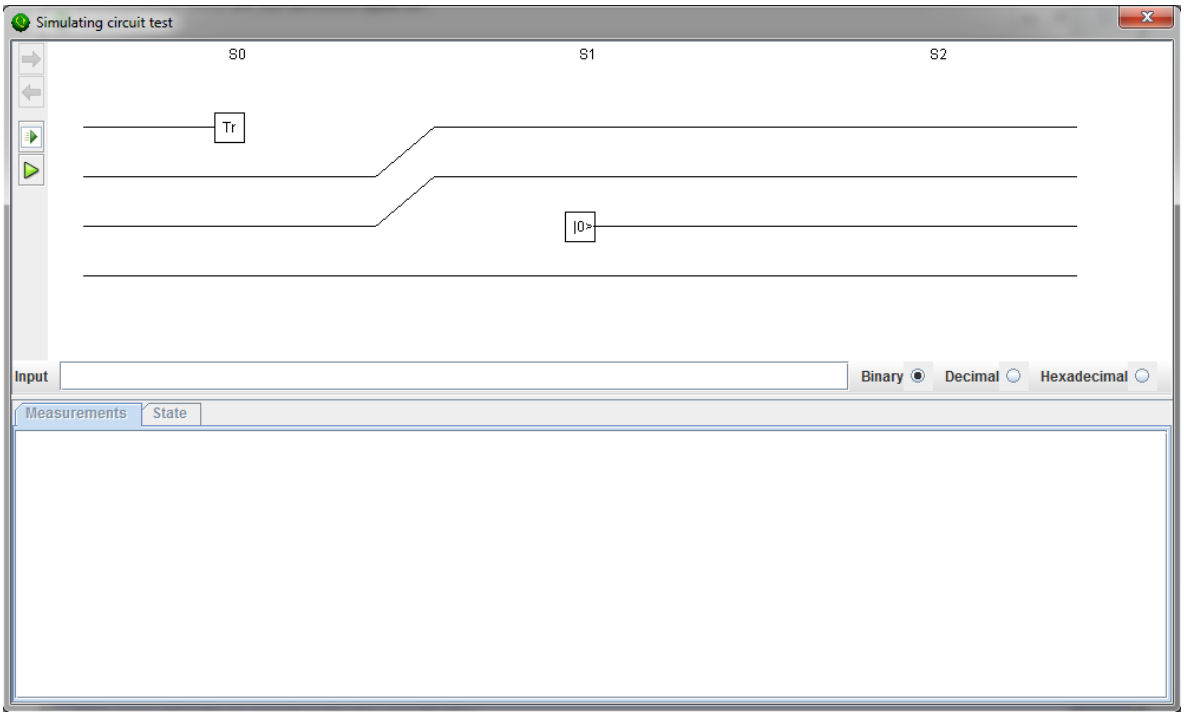
Además, el circuito seleccionado puede crear o eliminar qubits en su interior y, por tanto, puede tener diferente número de qubits a la entrada y a la salida, así que habrá que adecuar el tamaño de las etapas donde se vaya a añadir.

El contenido del circuito añadido se puede ver haciendo doble click sobre él, lo que abrirá una nueva ventana con su contenido. Se debe tener en cuenta que lo que se abre es una copia del circuito anterior, así que si se guarda no se guardará en el interior del circuito contenedor sino en un archivo aparte.

Por último, el circuito añadido se puede marcar como controlado tal y como si fuera una puerta común marcando la casilla correspondiente. Los qubits de control de su misma etapa controlarán a todas las puertas unitarias contenidas en él.

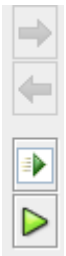
4.1.4 Simulando los circuitos

La ventana de simulación se abre pulsando el botón  :



Si el circuito tiene errores de diseño la ventana no se abrirá.

Esta ventana muestra el circuito a simular en el marco principal, una barra de herramientas en la parte izquierda:



Una barra de entrada en la que se indica el estado inicial en binario, decimal o hexadecimal según la casilla que se marque:

Y la zona donde se mostrarán los resultados.

Existen dos tipos de simulación:

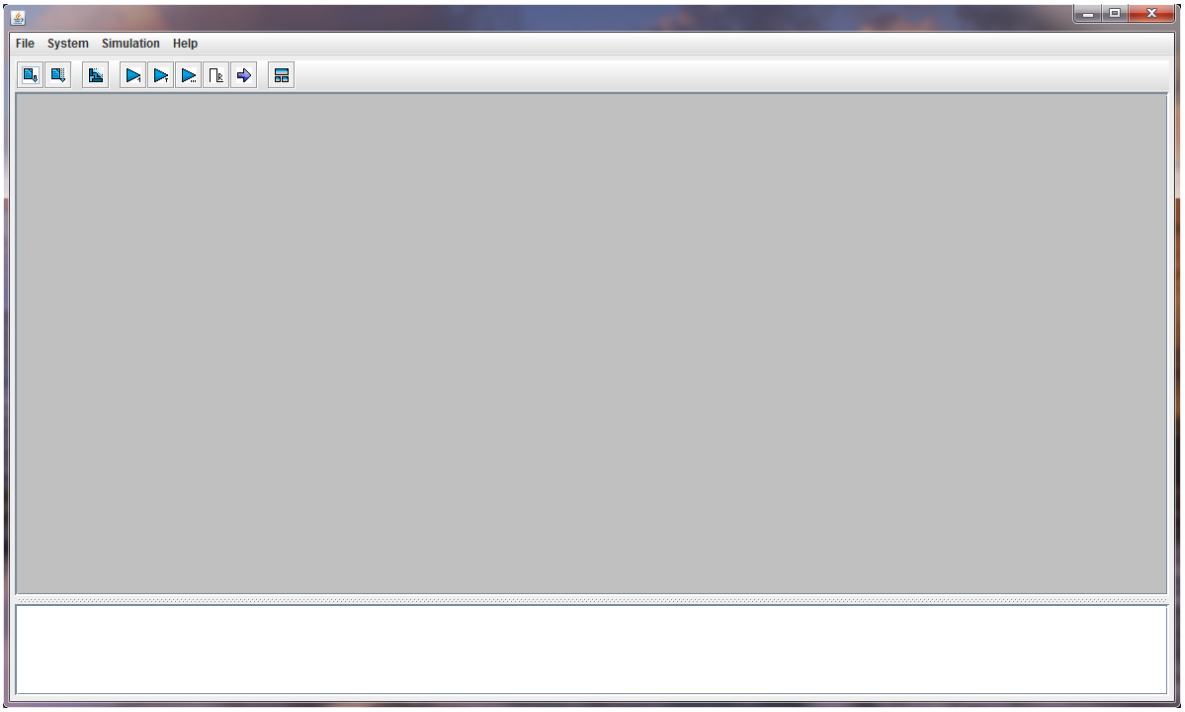
- : Realiza una simulación rápida, mostrando únicamente el resultado de las medidas.
- : Realiza una simulación detallada mostrando las medidas realizadas así como la evolución del estado etapa a etapa.

Se debe tener en cuenta que si el estado es muy grande en alguna etapa al sistema le puede resultar muy complicado imprimirlo por pantalla y un estado tan grande rara vez es representativo para el usuario. En estos casos se recomienda utilizar la simulación rápida.

Una vez simulado el sistema los resultados aparecerán en las pestañas inferiores, particularizados para cada etapa. Nos podemos desplazar por las etapas con los botones y y la etapa seleccionada se irá mostrando en gris.

4.2 Simulador del procesador cuántico qMIPS

La ventana principal del simulador QMIPS es la siguiente:



Tiene un menú superior:

Una barra de herramientas:

Una consola de información:

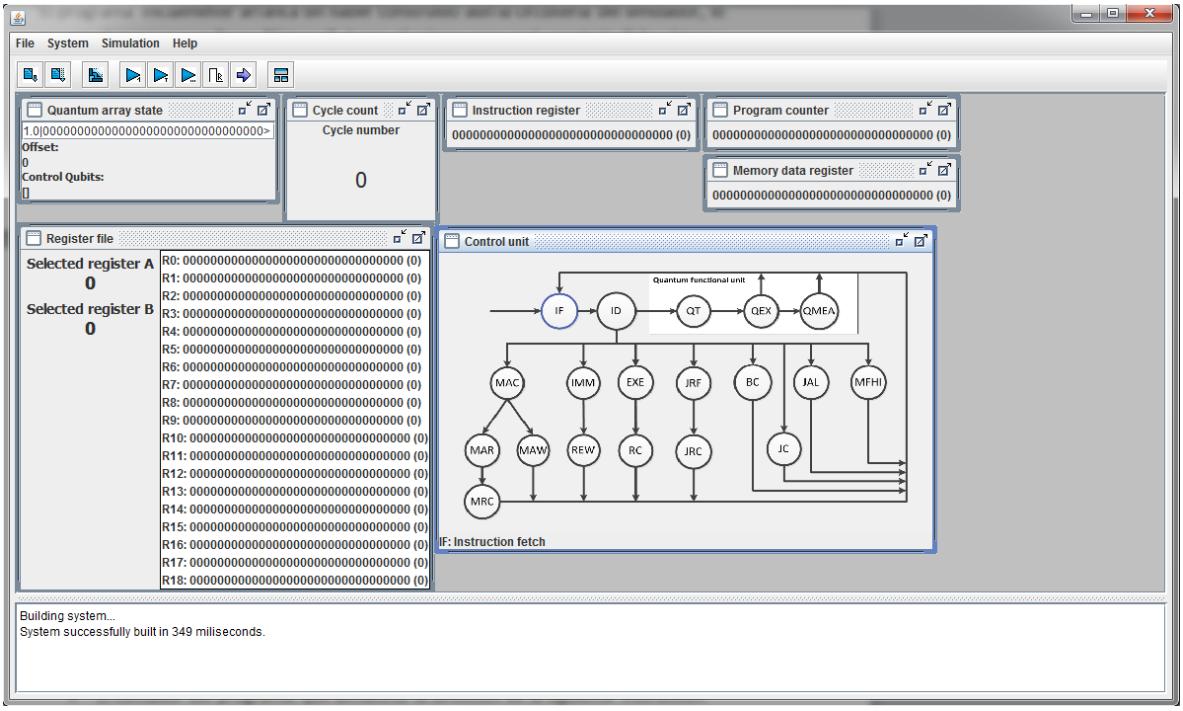
Y el espacio de trabajo.

El programa inicialmente arranca sin haber construido aun la circuitería del simulador, lo que en el programa se llama “Sistema”, luego el primer paso será construir dicho sistema pulsando el botón  o el ítem del menú “*System -> Build*”. El simulador construirá el

sistema e informará por la consola de información si todo ha ido bien. En ese momento mostrará en la ventana de trabajo todos los dispositivos relevantes para la ejecución.

4.2.1 Las vistas de los dispositivos

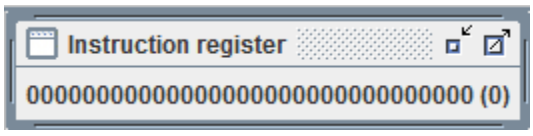
Cada dispositivo relevante tiene su propia vista de información:



Se muestran una serie de registros importantes del sistema:

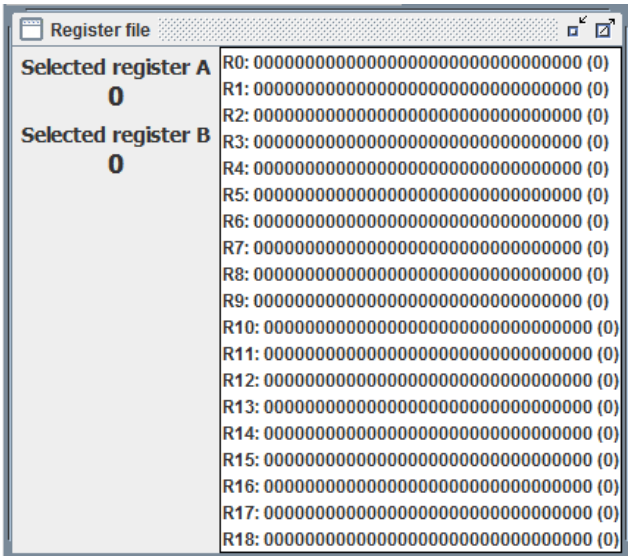
- El registro de instrucción: que guarda temporalmente la instrucción que se ha traído de memoria.
- El contador del programa: que almacena la dirección de la siguiente instrucción que se debe traer de memoria.
- El registro de dato de memoria: que almacena temporalmente el dato que se trae de la memoria.

Los registros se muestran de la siguiente forma:

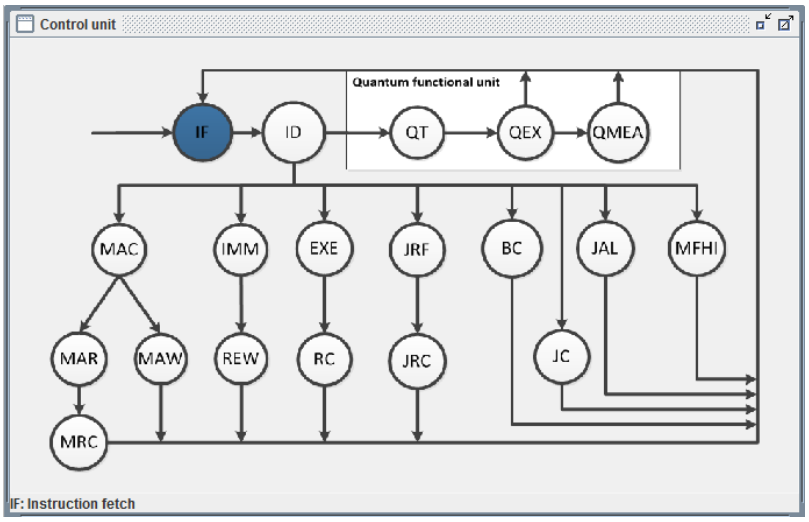


El valor del registro aparece en binario y entre paréntesis en decimal.

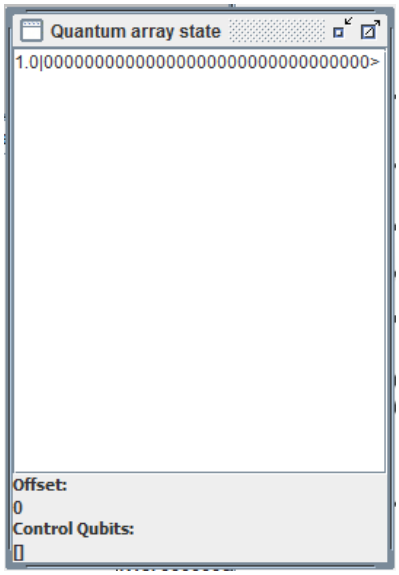
El fichero de registros tiene su propia vista que muestra el contenido de cada registro así como que registro está seleccionado a cada momento para leer:



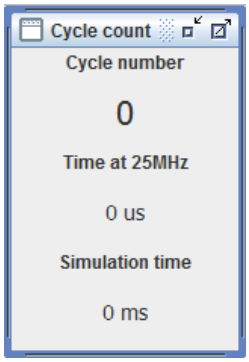
Para la unidad de control se muestra el árbol de estados completo. El estado actual del dispositivo aparecerá resaltado y señalado en la barra inferior de forma que sea sencillo seguir la ejecución de las instrucciones:



La unidad funcional cuántica muestra en un listado las componentes en superposición del estado interno de la unidad así como el *offset* actual y los qubits que están marcados como de control:



Por último, se dispone de una pequeña ventana que muestra el número de ciclos que se han ejecutado, el tiempo que habría necesitado el procesador MIPS I R3000 real a 25MHz (en µs) y el tiempo de simulación real (“*world clock time*”) medido mediante la función *System.currentTimeMillis()* (en ms), este tiempo puede ser algo más corto al real ya que no tiene en cuenta el repintado de la interfaz gráfica:



Las ventanas se pueden organizar automáticamente por la pantalla utilizando el botón . Si las ventanas no caben en la pantalla el programa las organizará en cascada para que el usuario las ubique como desee.

4.2.2 Cargando el código fuente

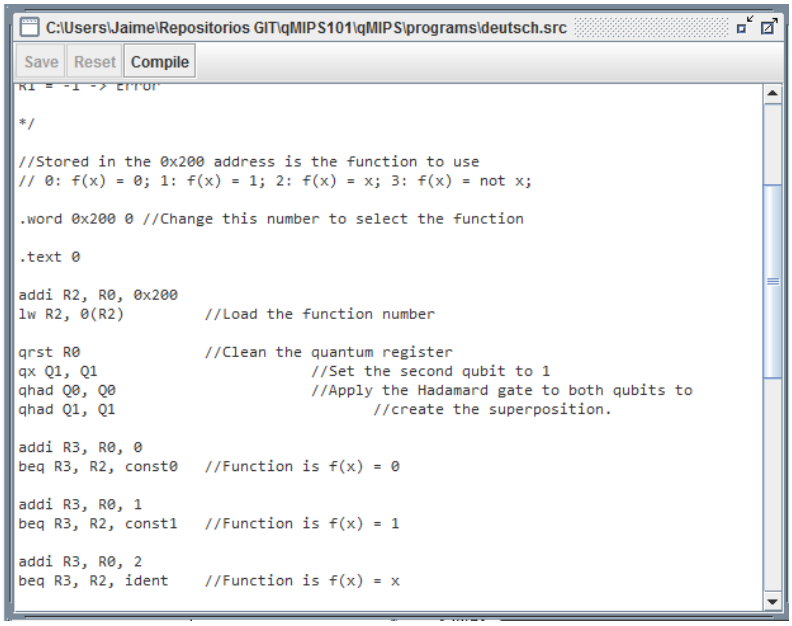
La memoria ahora mismo está vacía así que se debe cargar algún código fuente para que el simulador se ejecute. Para ello se pulsa el botón o el ítem del menú “*File -> Load*

source…". Podemos ejecutar el paso anterior y este de una vez utilizando el botón  o el ítem del menú "*File -> Build and load source…".*

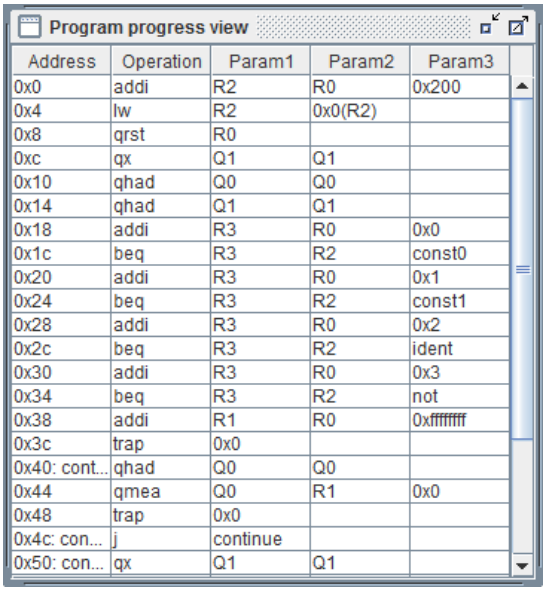
Ahora el programa intentará compilar el código fuente a la memoria, si se produce algún error, el programa mostrará los errores en la consola de información y seguirá adelante, pero el programa podría no haberse cargado correctamente.

Ahora se mostrarán dos nuevas ventanas:

- La ventana de edición de código fuente: muestra el código fuente tal y como fue escrito. Permite editar el código directamente, guardarlo con el botón  y recompilar el código guardado con el botón  . Además permite revertir las modificaciones realizadas hasta el documento guardado con el botón  .



- La vista de progreso del programa: muestra una tabla con la dirección la operación y los parámetros de cada instrucción del código, resaltando la instrucción que se está ejecutando en cada momento para que sea más fácil seguir la evolución del programa.



Program progress view

Address	Operation	Param1	Param2	Param3
0x0	addi	R2	R0	0x200
0x4	lw	R2	0x0(R2)	
0x8	qrst	R0		
0xc	qx	Q1	Q1	
0x10	qhad	Q0	Q0	
0x14	qhad	Q1	Q1	
0x18	addi	R3	R0	0x0
0x1c	beq	R3	R2	const0
0x20	addi	R3	R0	0x1
0x24	beq	R3	R2	const1
0x28	addi	R3	R0	0x2
0x2c	beq	R3	R2	ident
0x30	addi	R3	R0	0x3
0x34	beq	R3	R2	not
0x38	addi	R1	R0	0xffffffff
0x3c	trap	0x0		
0x40: cont...	qhad	Q0	Q0	
0x44	qmea	Q0	R1	0x0
0x48	trap	0x0		
0x4c: con...	j	continue		
0x50: con...	qx	Q1	Q1	

4.2.3 Simulando el sistema

El programa permite simular el comportamiento del procesador utilizando los tres botones de la barra de herramientas siguientes:

- El botón ejecuta un ciclo del programa cargado en memoria.
- El botón ejecuta el programa hasta que encuentra una excepción de programa. Típicamente se ejecuta hasta que se alcanza la instrucción “trap 0” que suele indicar el final del programa.
- El botón muestra una pequeña ventana desplegable donde podemos indicarle el número de ciclos que queremos que ejecute.

Cada vez que se ejecute un ciclo la información de cada ventana se modificará según vaya cambiando la información del sistema en tiempo real.

Si queremos reiniciar el sistema, pulsaremos el botón que dispara una señal de reset.

Durante la ejecución del programa pueden lanzarse varias excepciones, si esto ocurre el programa se bloquea en la excepción correspondiente hasta que el usuario pulsa el botón . Esto permite colocar varias instrucciones “trap 0” a modo de *puntos de interrupción* y ejecutar el programa de excepción en excepción.

5 Implementación de algoritmos cuánticos

Para comprobar el correcto funcionamiento del simulador, se han implementado dos de los algoritmos cuánticos más famosos en código ensamblador del qMIPS.

5.1 El algoritmo de Deutsch

El algoritmo de Deutsch fue descrito en detalle en la sección 2.3.1. La forma concreta de la implementación del algoritmo se ha obtenido de [1].

El algoritmo está implementado de forma que dependiendo de un índice en memoria llama a uno u otro oráculo. El resultado se almacenará en el registro R1 de la forma:

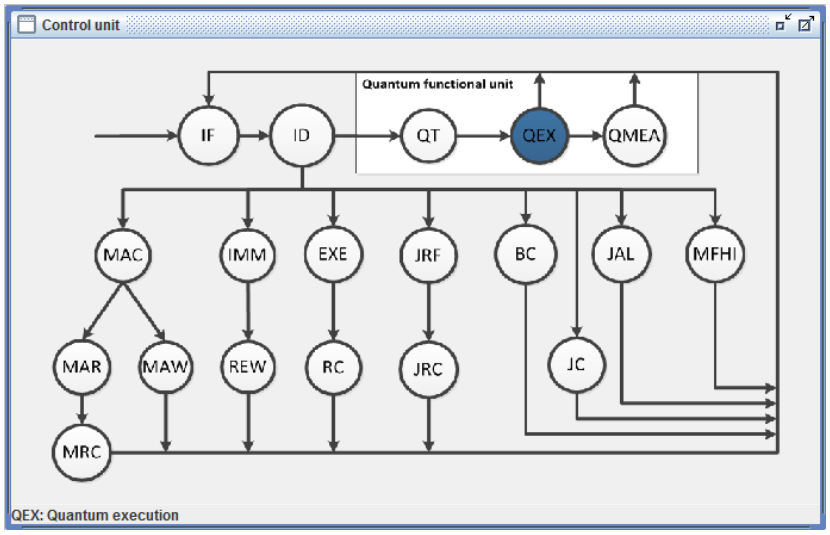
- R1 = 0: El oráculo es de tipo constante, es decir, $f(x) = 0$ o $f(x) = 1$.
- R1 = 1: El oráculo es de tipo equilibrado, es decir, $f(x) = x$ o $f(x) = not\ x$.
- R1 = -1: El índice de memoria no corresponde a ningún oráculo.

El primer paso será definir dicha posición de memoria y cargar el valor del índice del oráculo, siendo:

- [0x200] = 0, función $f(x) = 0$.
- [0x200] = 1, función $f(x) = 1$.
- [0x200] = 2, función $f(x) = x$.
- [0x200] = 3, función $f(x) = \bar{x}$.

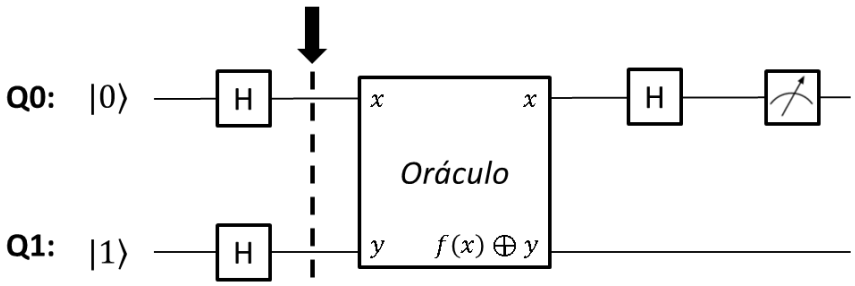
```
.word 0x200 1        //El número de la posición 200 indicará que oráculo
                     // se utiliza

.text 0
addi R2, R0, 0x200
lw R2, 0(R2)         //Se carga el índice de memoria.

qrst R0              //Limpiamos el registro cuántico
qx Q1, Q1            //Inicializamos el Segundo qubit a 1
qhad Q0, Q0          //Aplicamos la puerta de Hadamard a ambos qubits
qhad Q1, Q1          // para crear la superposición.
```

Podemos ver como el la unidad de control modifica el estado al pasar por la etapa QEX:

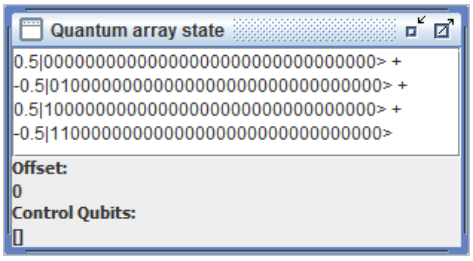


El programa está imitando, básicamente, el circuito cuántico que representa el algoritmo de Deutsch.

Hasta este momento hemos ejecutado:



Y el estado de la unidad cuántica en este punto es:



El siguiente paso sería realizar la llamada al oráculo correspondiente:

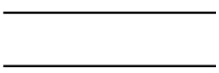
```
addi R3, R0, 0
beq R3, R2, const0   //El oráculo es f(x) = 0
```

```
addi R3, R0, 1
beq R3, R2, const1   //El oráculo es f(x) = 1

addi R3, R0, 2
beq R3, R2, ident    //El oráculo es f(x) = x

addi R3, R0, 3
beq R3, R2, not      //El oráculo es f(x) = not x

addi R1, R0, -1      //No existe una función con ese índice.
                     // devuelve error
trap 0
```

Es obvio que la subrutina correspondiente y por tanto el oráculo solo se ejecutará una vez.

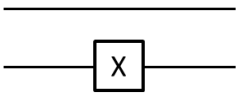
Ahora existen una serie de subrutinas que ejecutan cada oráculo:

- Para la función $f(x) = 0$, la subrutina no tiene que hacer nada ya que el resultado será siempre cero y no tendrá que invertir el segundo qubit nunca:

Índice	Código	Circuito
0	`const0:` `j continue`	

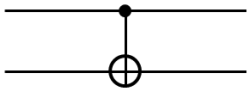
- Para la función $f(x) = 1$, la subrutina debe invertir el qubit inferior siempre ya que el resultado es siempre 1:

Índice	Código	Circuito

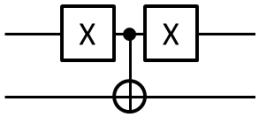
1	`const1:`
	`qx Q1, Q1`
	`j continue`
		

- En la función $f(x) = x$, habrá que invertir el qubit inferior si el superior vale 1, es decir, la operación CNOT:

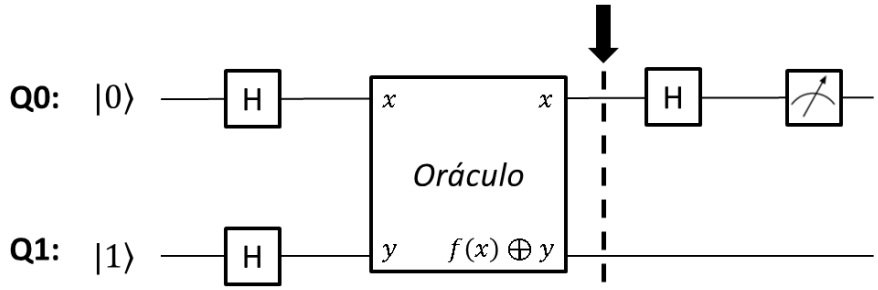
Índice	Código	Circuito

2	`ident:`
	`qx Q1, Q0`
	`j continue`
		

- La función $f(x) = not\ x$, tendrá que invertir el qubit inferior, si el superior vale cero. Esto se puede conseguir negando el qubit superior, ejecutando un CNOT y volviendo a negarlo para que vuelva a su estado original:

Índice	Código	Circuito
3	not: qx Q0, Q0 qx Q1, Q0 qx Q0, Q0 j continue	

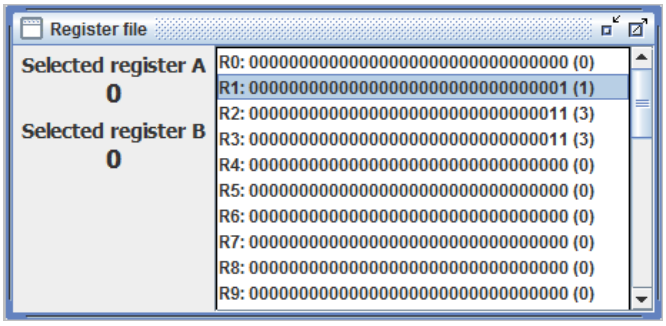
Hemos ejecutado hasta ahora:



Solo queda ejecutar la última puerta de Hadamard y realizar la medida para obtener el resultado:

```
continue:
//La puerta de Hadamard final
qhad Q0, Q0
//Volcamos la medida del primer qubit sobre el registro R1
qmea Q0, R1, 0
trap 0
```

En el registro R1 quedará el resultado de la forma antes descrita con tan sólo una llamada al oráculo. Por ejemplo para el oráculo $f(x) = not\ x$:



Como podemos observar, al medir el qubit Q1 hemos obtenido un 1 luego el oráculo que hemos utilizado debe ser por fuerza de tipo equilibrado.

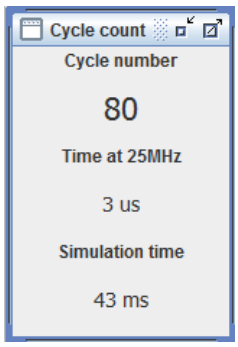
Se puede ver en el código que el programa tan solo hace una llamada a la subrutina oráculo, cuando con un computador clásico el único procedimiento posible para obtener el mismo resultado es el siguiente:

1. Llamamos al oráculo con un valor de entrada de 0, obteniendo $f(0) = a$. Tan solo con este valor es imposible saber si la función es constante o equilibrada.
2. Llamamos al oráculo con un valor de entrada de 1, obteniendo $f(1) = b$.
3. Ya tenemos información suficiente para saber si la función es constante o equilibrada: será constante si $a = b$ y equilibrada si $a \neq b$.

En este caso clásico, con tan solo una llamada al oráculo no disponemos de información suficiente para dar una respuesta. El computador cuántico tiene la capacidad de ejecutar funciones sobre estados del tipo $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$, evaluando ambos valores de la función al mismo tiempo, con lo que con tan solo una llamada es capaz de obtener la respuesta.

Por tanto, tenemos un algoritmo que el procesador qMIPS, como procesador cuántico, puede ejecutar de una forma que ningún procesador clásico podría.

El algoritmo se ejecuta en entre 48 y 80 ciclos de simulación dependiendo del oráculo y en un tiempo real del orden de los 40ms.



5.2 El algoritmo de Grover

El algoritmo de búsqueda de Grover fue descrito en detalle en la sección 2.3.1. Esta implementación se ha obtenido de [9].

En esta implementación concreta el algoritmo buscará simplemente un número concreto dentro de la colección de los número del 0 al 31 (5 qubits). En la dirección 0x600 de memoria se almacena el número que el algoritmo debe buscar y el oráculo correspondiente se genera automáticamente.

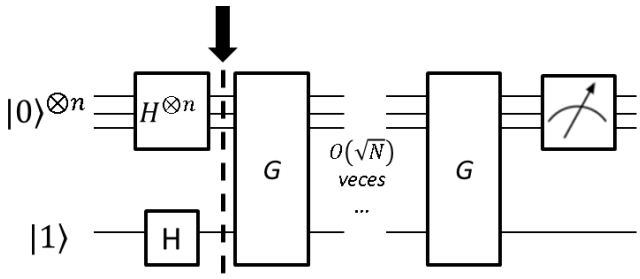
El primer paso es obtener el número a buscar de memoria:

```
//Este es el número que el algoritmo buscará en la colección
.word 0x600 7

.text 0

//Buscamos el dato en memoria
addi R5, R5, 0x600
lw R5, 0(R5)
```

La primera parte del algoritmo de Grover es colocar los qubits del espacio de búsqueda en superposición y aplicar una puerta de Hadamard a un qubit auxiliar en el estado $|1\rangle$:

```
//Limpiamos el registro cuántico.
qrst R0

//Aplicamos la puerta de Hadamard a los 5 qubits del subespacio de
// búsqueda para colocarlos en superposición.
qhad Q0, Q0
qhad Q1, Q1
qhad Q2, Q2
qhad Q3, Q3
qhad Q4, Q4

//Ponemos el qubit auxuliar en el estado correspondiente
qx Q5, Q5
qhad Q5, Q5
```

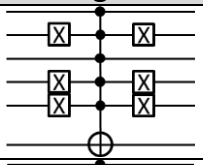
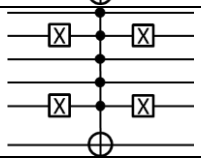
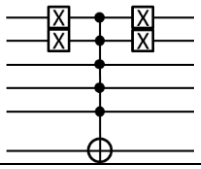
Ya hemos inicializado el sistema para empezar a ejecutar el operador de Grover, estamos en el punto:



El programa debe saber cuántas iteraciones del ciclo del Grover son necesarias, se puede demostrar que para 32 datos hay que ejecutar $\left\lceil \frac{\pi}{4}\sqrt{32} \right\rceil = 5$ iteraciones del operador de Grover:

```
//Contador de iteraciones a 5 en R2
addi R2, R0, 5

grover:

//Llamamos a la subrutina oracle que genera el oráculo correspondiente al
// número elegido.
jal oracle

//Ejecutamos la inversión sobre la media: Hxn(Cambio de Fase)Hxn
qhad Q0, Q0
qhad Q1, Q1
qhad Q2, Q2
qhad Q3, Q3
qhad Q4, Q4

//La subrutina phase realiza el cambio de fase del operador
jal phase

qhad Q0, Q0
qhad Q1, Q1
qhad Q2, Q2
qhad Q3, Q3
qhad Q4, Q4

//Reducimos el contador de iteraciones y saltamos a la siguiente si no
// hemos acabado
addi R2, R2, -1
bne R2, R0, grover
```

La subrutina *oracle* genera un circuito que niega el qubit auxiliar si los otros cinco tienen el valor binario buscado, por ejemplo (el qubit superior es el menos significativo):

Valor decimal	Valor binario	Circuito generado
5	00101	
13	01101	
28	11100	

Funciona de la siguiente forma:

- Se mantiene en R7 un contador de desplazamientos, en R9 se almacena un 1 para utilizarlo de máscara y R8 guarda el número total de desplazamientos:

```
oracle:
addi R8, R0, 5
addi R9, R0, 1
add R7, R0, R0
```

- Se va desplazando el valor binario buscado a la derecha R7 veces y se enmascara con 1 para quedarnos con el bit menos significativo. Si este bit es cero, debemos ejecutar una puerta X sobre el qubit apuntado por R7. Al terminar deshacemos los cambios:

```
setnot:
srl R6, R5, R7
and R6, R6, R9
bne R6, R0, setisone
qoff R7
qx Q0, Q0
setisone:
addi R7, R7, 1
bne R7, R8, setnot

qoff R0
add R7, R0, R0
```

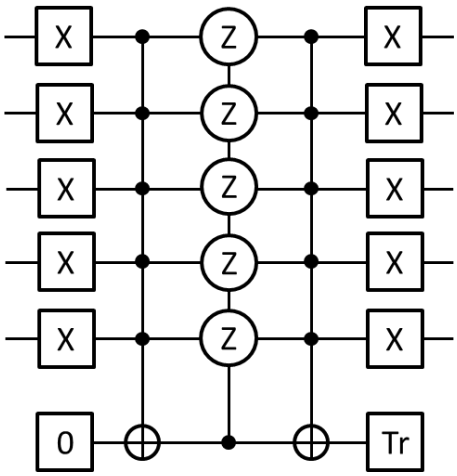
- Ahora se marcan todos los qubits menos el auxiliar como qubits de control, y se ejecuta la puerta X controlada por todos ellos sobre el auxiliar:

```
setcontrol:
qcnt R7
addi R7, R7, 1
bne R7, R8, setcontrol

qx Q5, Q5
```

- Finalmente, se repiten los pasos anteriores para que los qubits afectados dejen de ser de control y volver a aplicar la puerta X a los que la tuvieran:

```
add R7, R0, R0

unsetcontrol:
qcnt R7
addi R7, R7, 1
bne R7, R8, unsetcontrol

add R7, R0, R0

unsetnot:
srl R6, R5, R7
and R6, R6, R9
bne R6, R0, unsetisone
qoff R7
qx Q0, Q0
unsetisone:
addi R7, R7, 1
bne R7, R8, unsetnot

endgrover:
qoff R0
jr R31
```

La subrutina de cambio de fase (*phase*) ejecuta simplemente el siguiente circuito:

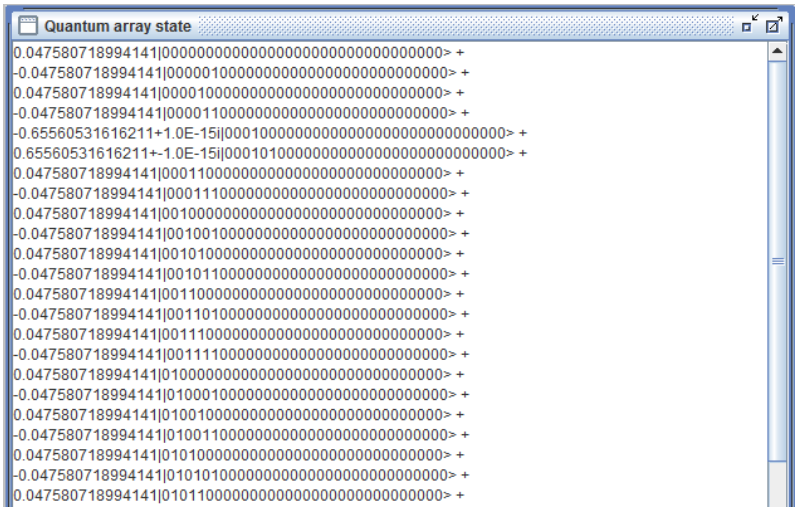


Que en código se representa como:

```
phase:

qx Q0, Q0
qx Q1, Q1
qx Q2, Q2
qx Q3, Q3
qx Q4, Q4

addi R20, R0, 0
qcnt R20
addi R20, R0, 1
qcnt R20
addi R20, R0, 2
qcnt R20
addi R20, R0, 3
qcnt R20
addi R20, R0, 4
qcnt R20

qx Q6, Q6

addi R20, R0, 1
qphs Q0, Q6, R20
qphs Q1, Q6, R20
qphs Q2, Q6, R20
qphs Q3, Q6, R20
qphs Q4, Q6, R20

qx Q6, Q6

addi R20, R0, 0
qcnt R20
addi R20, R0, 1
qcnt R20
```

```
addi R20, R0, 2
qcnt R20
addi R20, R0, 3
qcnt R20
addi R20, R0, 4
qcnt R20

qx Q0, Q0
qx Q1, Q1
qx Q2, Q2
qx Q3, Q3
qx Q4, Q4

qmea Q6, R20, 0
beq R20, R0, endphase
qx Q6, Q6


endphase:
jr R31
```

Tras las iteraciones correspondientes el estado queda:



Vemos que la probabilidad de obtener $|00010\rangle = |8\rangle$, si elegimos como oráculo ese número, es mucho más alta que las del resto de las componentes:

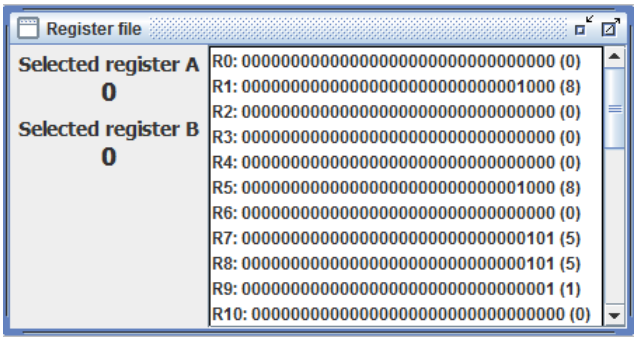
$$probabilidad\ de\ 8 = |-0.6556i|^2 + |0.6556i|^2 = 0.8596\ (85\%)$$

$$probabilidad\ de\ no\ 8 = 1 - 0.8596 = 0.14\ (14\%)$$

Tan solo con medir dichos qubits sobre el registro R1 obtendremos el resultado esperado con dicha probabilidad:

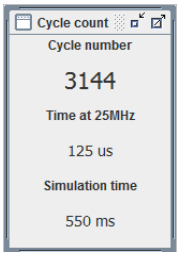
```
qmea Q0, R1, 0
qmea Q1, R2, 1
or R1, R1, R2
qmea Q2, R2, 2
or R1, R1, R2
qmea Q3, R2, 3
or R1, R1, R2
qmea Q4, R2, 4
or R1, R1, R2

trap 0
```

Quedando en el fichero de registros:



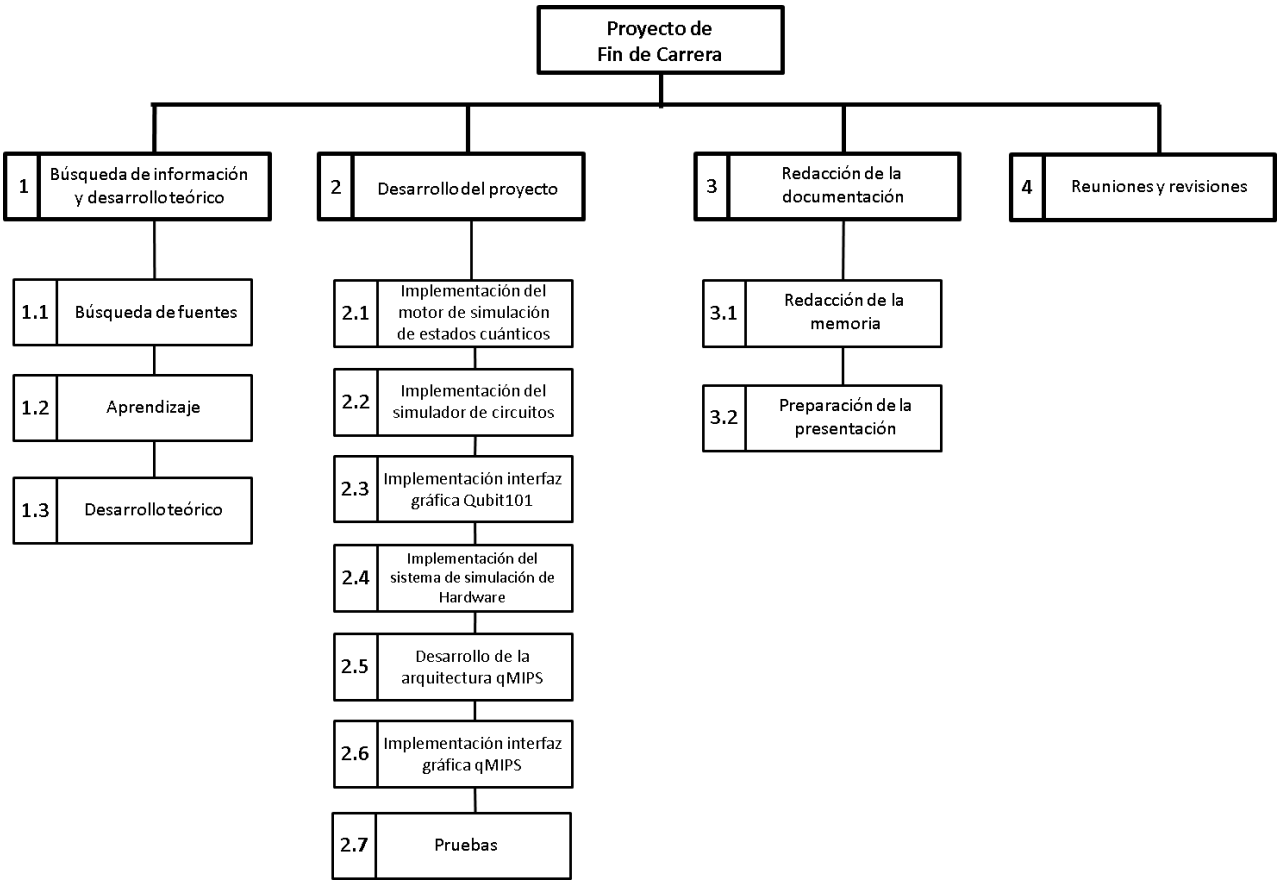
La potencia de este algoritmo está en que, con la única capacidad de identificar un número concreto (el 8 en el caso del ejemplo) entre los valores del 0 al 31, ha sido capaz de encontrar dicho número en tan solo 5 intentos. Como ejemplo clásico, es como si se colocaran 32 cartas bocabajo sobre una mesa y se nos pidiera encontrar la carta con el número 8. Si están desordenadas, la única estrategia posibles ir probando de carta en carta hasta dar con la buscada, lo que de media serán 32/2 = 16 intentos. Como hemos visto el algoritmo de búsqueda de Grover lo consigue con una alta probabilidad en 5 intentos.

El algoritmo tarda aproximadamente medio segundo en ejecutarse por completo simulando 3144 ciclos. Si se ejecutara en el qMIPS real a 25MHz le llevaría 125µs:



6 Análisis temporal y de costes de desarrollo

El proyecto de fin de carrera se puede dividir en las siguientes tareas:



A continuación se desglosan cada una de ellas en mayor detalle, con una estimación del esfuerzo (en *persona-día,* teniendo en cuenta 8 horas de trabajo al día) y se realiza un análisis temporal y de costes totales del proyecto.

6.1 Búsqueda de información y desarrollo teórico

Se refiere a la preparación personal para poder realizar correctamente el proyecto, debido que existen partes de este que quedan fuera del alcance de la formación recibida durante la carrera. Se puede desglosar a su vez en las siguientes tareas:

1. **Búsqueda de fuentes:** Localizar tanto literatura como artículos relevantes a la materia tanto del desarrollo del proyecto como de la documentación (*3 personas-día*).
2. **Aprendizaje:** Formación personal en el campo correspondiente, de forma que se tenga un conocimiento los más completo posible del campo al que va dirigido el proyecto (*1 persona-día*).
3. **Desarrollo teórico:** Estudio teórico de la viabilidad de cada una de las partes del proyecto (*1 persona-día*).

El esfuerzo total de desarrollo de esta tarea es, por tanto, de *5 personas-día*.

6.2 Desarrollo del proyecto

El proyecto está construido por módulos dependientes unos de otros, pero con un acoplamiento mínimo. De esta forma, es posible seguir un desarrollo iterativo de cada uno de ellos, al menos a partir de la primera versión.

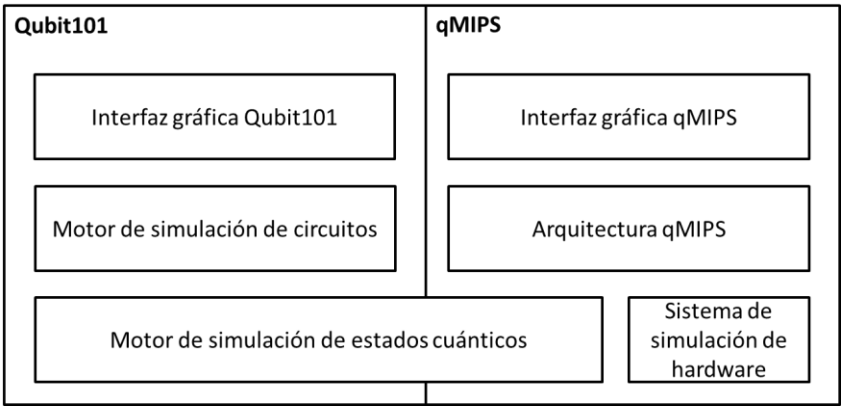
Consta de dos aplicaciones finales, el simulador de circuitos cuánticos Qubit101 y el simulador del procesador cuántico qMIPS. Esta última depende de la primera en cuanto a que utiliza su motor de simulación. El desarrollo de ambas herramientas se puede desglosar en las siguientes tareas:

1. **Implementación del motor de simulación de estados cuánticos:** Desarrollo de este motor de simulación que utilizarán ambas herramientas para simular los estados (*9 personas-día*).
2. **Implementación del motor de simulación de circuitos:** Implementación de las estructuras de datos que imiten el comportamiento de los circuitos cuánticos. Esta tarea se apoya en la anterior (*4 personas-día*).
3. **Implementación de la interfaz gráfica del simulador Qubit101:** A partir del motor anterior, proporcionar a los usuarios una interfaz ágil para desarrollar circuitos cuánticos (*6 personas-día*).
4. **Implementación del sistema de simulación de hardware:** Tarea independiente que en la que se desarrolla un sistema que permite simular cualquier hardware en Java de forma realista y versátil (*9 personas-día*).
5. **Desarrollo de la arquitectura qMIPS:** Desarrollo partiendo del procesador presentado en [2], de la arquitectura del procesador cuántico y su implementación en el sistema de la tarea anterior (6 *personas-día*).

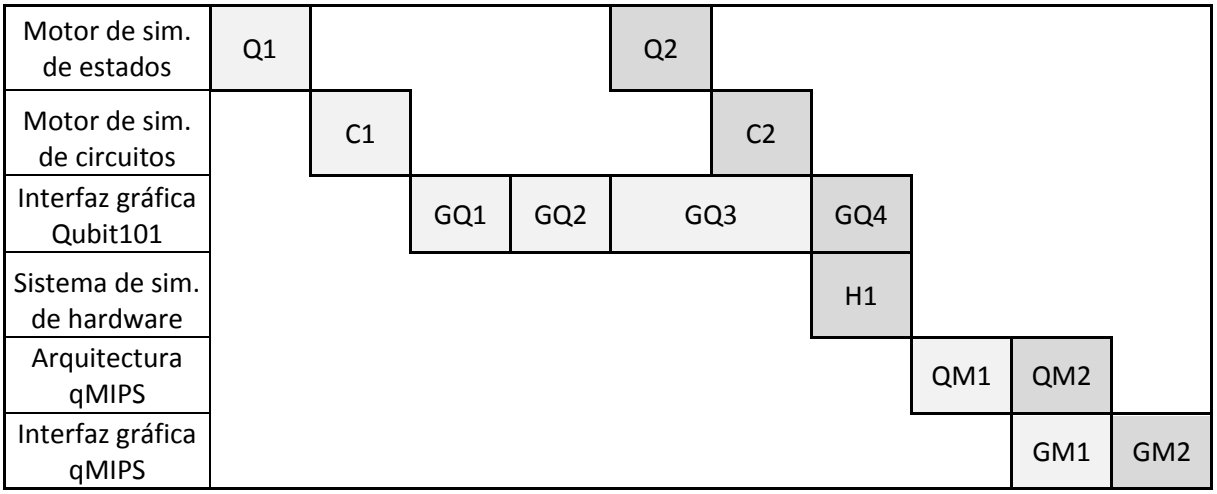
6. **Implementación de la interfaz gráfica del simulador qMIPS:** Se proporciona a los usuarios una interfaz gráfica que de toda la información relevante del estado del procesador simulado ciclo a ciclo (*6 personas-día*).
7. **Pruebas:** Pruebas exhaustivas a ambas herramientas, desarrollando y probando diferentes circuitos para Qubit101 y programas para qMIPS (*4 personas-día*).

Tenemos ahora un esfuerzo total estimado para esta tarea de *44 personas-día*.

La estructura general de las aplicaciones se puede ver fácilmente en el siguiente esquema:



Como se ha dicho anteriormente en cada parte del sistema se ha seguido un desarrollo iterativo. El siguiente diagrama muestra las diferentes versiones por las que ha pasado cada componente:



Donde cada uno representa:

- Q1: Motor matricial de estados cuánticos.
- Q2: Motor de mapa de estados (actual).
- C1: Primera versión del simulador de circuitos.
- C2: Versión mejorada (actual).
- GQ1: Interfaz gráfica de tipo consola del simulador.
- GQ2: Versión gráfica básica utilizando AWT.
- GQ3: Primera versión SWING.
- GQ4: Versión avanzada con SWING (actual).
- H1: única versión del sistema de simulación de hardware.
- QM1: Versión básica del procesador, tal y como aparecía en [2].
- QM2: Versión extendida del procesador, ya con instrucciones cuánticas (actual).
- GM1: Versión básica de la interfaz gráfica del simulador.
- GM2: Versión avanzada de la interfaz (actual).

6.3 Redacción de la documentación

Esta tarea se refiere a la redacción y preparación de toda la documentación requerida y generada por el proyecto. Se divide en dos subtareas:

1. **Redacción de la memoria:** Redacción, maquetación y preparación de esta memoria (*9 personas-día*).
2. **Preparación de la presentación:** Diseño de las diapositivas y guiones, así como los ensayos de la presentación del proyecto (*3 personas-día*).

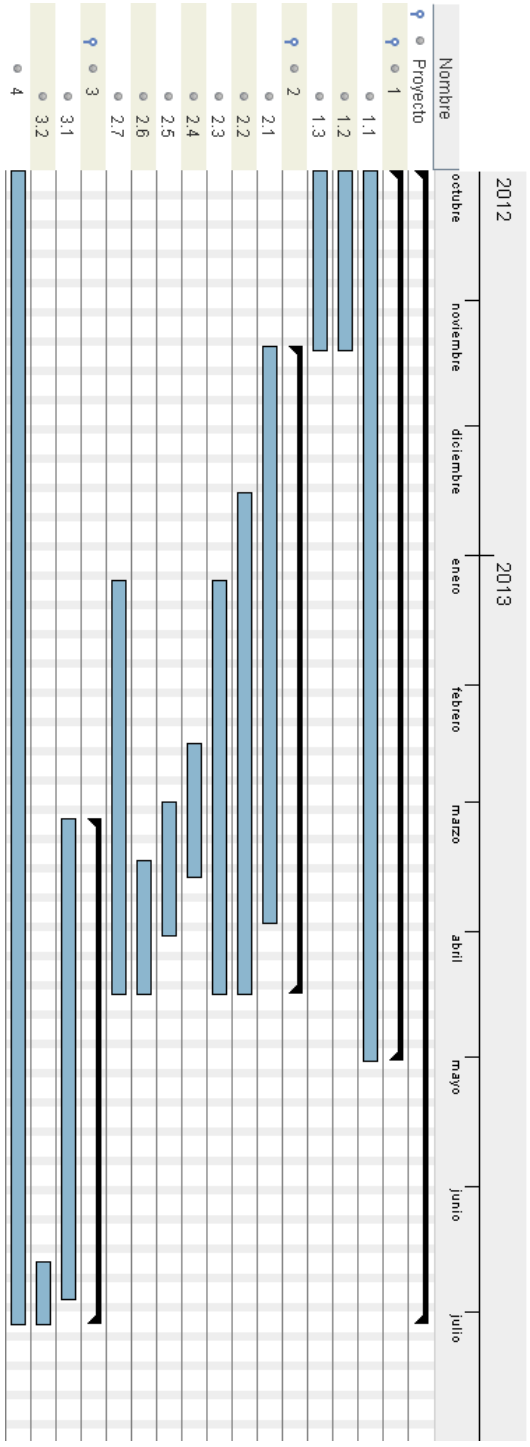
El esfuerzo total es aquí de *12 personas-día.*

6.4 Reuniones y revisiones

Por último, hay que tener en cuenta el tiempo de reuniones y revisiones de la documentación, la presentación y el software del proyecto, tanto por parte del alumno como del tutor (*4 personas-día*).

6.5 Análisis temporal y de costes totales

A continuación se presenta una estimación de los tiempos requeridos para cada una de las tareas que se acaban de describir. El desarrollo temporal de las tareas ha sido tal y como muestra el diagrama siguiente [15]:



El esfuerzo total combinado de todas las tareas del proyecto sería, sumando el esfuerzo de cada subtarea, el siguiente:

$$\left.\begin{array}{c} Busqueda\ de\ información\ y\ desarrollo\ teórico = 5\ personas \cdot día \\ Desarrollo\ del\ proyecto = 44\ personas \cdot día \\ Redaccion\ de\ la\ documentación = 12\ personas \cdot día \\ Reuniones\ y\ revisiones = 4\ personas \cdot día \end{array}\right\} TOTAL = \ 65\ personas \cdot día$$

Es decir, obtenemos un esfuerzo total para el proyecto en personas-mes de:

$$\frac{65\ personas - día}{20\ días/mes} = \mathbf{3.25\ personas - mes}$$

Teniendo en cuenta un trabajo medio de 8 horas por día, tenemos una duración estimada del proyecto de 520 horas.

Si tenemos ahora en cuenta el coste medio de un ingeniero informático titulado de aproximadamente 20€ por hora, nos da un coste total del proyecto de:

$$520\ horas \cdot \frac{20€}{hora} = 10400€$$

Dado que todo el software que se ha utilizado es de licencia libre, no hay que añadir a este total ningún otro coste material.

7 Conclusiones

Se han desarrollado dos simuladores del campo de la computación cuántica, totalmente desarrollados con software libre.

El simulador qMIPS, integra en un procesador derivado del MIPS I clásico, una unidad funcional capaz de realizar computaciones cuánticas. Esto permite programar algoritmos cuánticos en lenguaje ensamblador, con la versatilidad que ello conlleva con respecto al modelo de circuitos, alcanzando de forma mucho más sencilla todo tipo de programas y convirtiendo a este procesador en un computador cuántico universal.

La herramienta permite:

- Cargar, modificar y guardar los programas.
- Simular el comportamiento del procesador ciclo a ciclo.
- Mostrar cada uno de los componentes internos relevantes del procesador clásico.
- Observar el estado de la unidad funcional cuántica, mostrando cómo evoluciona con el programa

No existe ningún otro simulador libre de este tipo, que implemente toda una arquitectura clásica con las capacidades de computación cuántica integradas, haciendo a este simulador una nueva herramienta con la que experimentar en el desarrollo de algoritmos cuánticos y su implementación física.

Las características de encapsulación que Java nos proporciona, permite que esta herramienta utilice el motor de simulación de estados cuánticos desarrollado para el segundo simulador que se presenta.

El simulador de circuitos cuánticos Qubit101 permite:

- Crear, modificar y guardar los circuitos cuánticos.
- Simular su efecto sobre un estado cuántico inicial definido por el usuario.
- Observar la evolución del estado etapa a etapa, junto con el resultado de las medidas que se hayan realizado.
- Utilizar otros circuitos cuánticos como puertas, pudiendo crear cómodamente circuitos de una complejidad indefinida.
- Simular un número prácticamente arbitrario de qubits, eso sí, manteniendo la superposición por debajo de 2^{22}componentes.

Con todo esto, esta herramienta supera a la mayoría de los simuladores libres del mismo tipo, facilitando la labor de los desarrolladores de algoritmos cuánticos que verán en ella un banco de trabajo donde poner a prueba sus desarrollos.

Ambos programas permitirán iniciar a los futuros ingenieros, al mismo tiempo, en el mundo de la computación cuántica y de la arquitectura de procesadores. De esta forma se puede mostrar desde el inicio de la formación la máquina clásica y la cuántica como un solo procesador combinado, lo que hará más sencillo comprender esta unión y aceptarla con naturalidad.

A las personas ya formadas en arquitectura de procesadores, el simulador qMIPS les ayudará a entender la computación cuántica desde una base que ya conocen, dado que la arquitectura clásica en la que se apoya es muy sencilla.

Al estar desarrollada en Java, la aplicación es fácil de extender y complementar. De esta forma, podrían desarrollarse siguiendo el guion de la herramienta nuevos simuladores de procesadores clásicos, por ejemplo con algún tipo de arquitectura paralela clásica; o de nuevos paradigmas de la computación, como computación biológica. De hecho, ambos proyectos están hospedados en la plataforma de desarrollo colaborativo de software GitHub [16], de forma que cualquier desarrollador interesado podría mejorar las herramientas [17].

8 Futuras líneas

El desarrollo en Java del proyecto, siguiendo la filosofía del acoplamiento mínimo, hace sencillo extenderla en cada uno de sus componentes sin alterar demasiado ninguno de los otros.

La integración de la unidad funcional cuántica al procesador nos abre una puerta a la simulación de otros efectos cuánticos importantes, por ejemplo, podría incluir un mecanismo que le permitiera simular la decoherencia del estado cuántico, por ejemplo ejecutando en un tiempo aleatorio alguna puerta cuántica que altere el estado. Así, se podrían desarrollar algoritmos que intenten evitar los efectos de la decoherencia agrupando las operaciones cuánticas o utilizando algoritmos de corrección de errores. Incluso se podría simular un canal de envío de información cuántica dejando el estado sin alterar durante varios ciclos, dejando que actúe tan solo la decoherencia.

Dado que el sistema de simulación de hardware es independiente de la arquitectura que se desarrolle sobre él, se podrían diseñar otras arquitecturas (reales o ficticias), como la versión encadenada del MIPS, y plasmarlas en algún lenguaje de descripción, como por ejemplo XML, de forma que la herramienta sea capaz de interpretarlo y generar cualquier sistema. Un paso más por encima sería desarrollar una herramienta gráfica que genere dicho código, de forma que sea más sencillo de construir.

Las interfaces gráficas de cada uno de los componentes del procesador en el simulador qMIPS son independientes de la interfaz general y pueden ser modificadas a voluntad. Por ejemplo, en la interfaz de la unidad funcional cuántica, se podría añadir una ventana que mostrara probabilidad de cada componente de la superposición en forma de gráfica, de forma que con un vistazo se pudiera observar el progreso del estado.

El compilador del que dispone la herramienta podría mejorarse para aceptar múltiples archivos de código que se pudieran utilizar como librerías de forma que se pudiera construir código por módulos pudiendo realizar así programas de mayor complejidad.

Por último, cabe señalar que dado que se trata de una herramienta de código libre [17], está abierta al cualquier desarrollo posterior por los programadores interesados.

9 Bibliografía

[1] Michael A. Nielsen and Isaac L. Chuang (2000), *Quantum Computation and Quantum Information*. Cambridge University Press (Cambridge).

[2] David A. Patterson and John L. Hennessy (2007), *Computer Organization and Design: The Hardware/Software Interface (3rd edition)*, Morgan Kaufmann Publishers Inc., San Francisco, CA.

[3] Xiao-Song Ma, Thomas Herbst, Thomas Scheidl,Daqing Wang, Sebastian Kropatschek, William Naylor, Bernhard Wittmann, Alexandra Mech, Johannes Kofler, Elena Anisimova, Vadim Makarov, Thomas Jennewein, Rupert Ursin & Anton Zeilinger (2012), *Quantum teleportation over 143 kilometres using active feed-forward*, Nature 489, 269–273.

[4] Thomas Monz, Philipp Schindler, Julio T. Barreiro, Michael Chwalla, Daniel Nigg, William A. Coish, Maximilian Harlander, Wolfgang Haensel, Markus Hennrich, Rainer Blatt (2011), *14-qubit entanglement: creation and coherence*, Phys. Rev. Lett. 106, 130506.

[5] Matteo Mariantoni, H. Wang, T. Yamamoto, M. Neeley, Radoslaw C. Bialczak, Y. Chen, M. Lenander, Erik Lucero, A. D. O'Connell, D. Sank, M. Weides, J. Wenner, Y. Yin, J. Zhao, A. N. Korotkov (2011), A. N. Cleland, John M. Martinis, *Implementing the Quantum von Neumann Architecture with Superconducting Circuits*, Science 334, 61-65.

[6] D. Deutsch (1985), *Quantum Theory, the Church-Turing Principle and the Universal Quantum Computer*, Proc. R. Soc. Lond. A 8 July 1985 vol. 400 no. 1818 97-117.

[7] Dominic Sweetman (2007), *See MIPS run*, San Francisco, Calif.: Morgan Kaufmann Publishers/Elsevier.

[8] Lov K. Grover (1996), *A fast quantum mechanical algorithm for database search*, Proceedings, 28th Annual ACM Symposium on the Theory of Computing (STOC), May 1996, pages 212-219.

[9] Eleanor Rieffel and Wolfgang Polak (2011), *Quantum computing: a gentle introduction*, Cambridge, MA [etc]: The MIT Press.

[10] Mikio Nakahara, Tetsuo Ohmi (2008), *Quantum computing: from linear algebra to physical realizations*, Boca Raton: CRC Press.

[11] Noson S. Yanofsky and Mirco A. Mannucci (2008), Quantum computing for computer scientists, New York, NY : Cambridge University Press.

[12] Dan C. Marinescu, Gabriela M. Marinescu (2011), *Classical and quantum information*, Amsterdam [etc] : Elsevier [etc].

[13] ANTLR (ANother Tool for Language Recognition), http://www.antlr.org/.

[14] R. Garcia-Patron Sanchez, J. Fiurasek, N.J. Cerf, J. Wenger, R. Tualle-Brouri, Ph. Grangier (2004), *Proposal for a loophole-free Bell test using homodyne detection*, Phys. Rev. Lett. 93, 130409.

[15] GanttProject 2.6, http://www.ganttproject.biz/.

[16] Sitio web oficial de GitHub, https://github.com/.

[17] Página del proyecto en GitHub, https://github.com/jaimecp89/qMIPS101.

[18] Lieven M. K. Vandersypen, Matthias Steffen, Gregory Breyta, Costantino S. Yannoni, Mark H. Sherwood & Isaac L. Chuang (2001), *Experimental realization of Shor's quantum factoring algorithm using nuclear magnetic resonance,* Nature 414, 883-887 (20 December 2001).

[19] J. I. Cirac and P. Zoller (1995), *Quantum Computations with Cold Trapped Ions,* Phys. Rev. Lett. 74, 4091–4094.

[20] C. H. Bennett, G. Brassard, *Quantum Cryptography: Public Key Distribution and Coin Tossing,* Proceedings of the IEEE International Conference on Computers, Systems and Signal Processing (1984), pp. 175-179.

[21] John S. Bell (1966), *On the Problem of Hidden Variables in Quantum Mechanics*, Rev. Mod. Phys. 38, 447–452.

[22] List of QC simulators, http://www.quantiki.org/wiki/List_of_QC_simulators.

[23] Kanamori, Y., Yoo, S., Pan, W. D., & Sheldon, F. T. (2006). *A short survey on quantum computers,* International Journal of Computers and Applications, 28(3), 227-233.

Anexo A: Código fuente

A.1 Proyecto qMIPS

Archivo *qMIPS / qmips.compiler.Analex.g*

```
header{
	package qmips.compiler;
}

class Analex extends Lexer;

options{
 k = 2;
 importVocab = Anasint;
 caseSensitive = false;
}

tokens{
  ADD = "add";
  ADDU = "addu";
  SUB = "sub";
  SUBU = "subu";
  ADDI = "addi";
  ADDIU = "addiu";
  MULT = "mult";
  DIV = "div";
  DIVU = "divu";
  LD = "ld";
  LW = "lw";
  LH = "lh";
  LHU = "lhu";
  LB = "lb";
  LBU = "lbu";
  SD = "sd";
  SW = "sw";
  SH = "sh";
  SB = "sb";
  LUI = "lui";
  MFHI = "mfhi";
  MFLO = "mflo";
  MFCZ = "mfcz";
  MTCZ = "mtcz";

  AND = "and";
  OR = "or";
  ORI = "ori";
  XOR = "xor";
  NOR = "nor";
  SLT = "slt";
  SLTI = "slti";
  SLL = "sll";
  SRL = "srl";
  SRA = "sra";

  BEQ = "beq";
  BNE = "bne";
  J = "j";
  JR = "jr";
  JAL = "jal";

  QHAD = "qhad";
  QX = "qx";
  QY = "qy";
  QZ = "qz";
  QMEA = "qmea";
  QPHS = "qphs";
  QNPH = "qnph";
  QRST = "qrst";
  QCNT = "qcnt";
  QOFF = "qoff";

  TRAP = "trap";
```

```
}

protected NEW_LINE: "\r\n"
{newline();};

BLANK: (' '|'\t' | "\r\n")
{$setType(Token.SKIP);};

LINECOM: "//" (~'\r')*
{$setType(Token.SKIP);};

BLOCKCOM: "/*" (options {greedy=false;}:(NEW_LINE|.))* "*/"
{$setType(Token.SKIP);};

protected DIGIT: '0'..'9';
protected CHAR: 'a'..'z';

NUMBER: DIGIT (DIGIT)*;
NUMDOUBLE: DIGIT "." (DIGIT)*;
HEXADECIMAL : "0x" (DIGIT | 'a'..'f') (DIGIT | 'a'..'f')*;
STRING: (CHAR|DIGIT)(CHAR|DIGIT)*;

MHEX: '-' HEXADECIMAL;
MDEC: '-' NUMBER;

R_REGISTER:  'r' NUMBER;
F_REGISTER:  'f' NUMBER;
D_REGISTER:  'd' NUMBER;
Q_REGISTER:  'q' NUMBER;

OP : '(';
CP : ')';
P  : '.';
C  : ',';
DP : ':';
OC : '[';
CC : ']';

 BYTE : ".byte";
 HWORD : ".hword";
 WORD : ".word";
 DOUBLE : ".double";

 TEXT : ".text";
```

Archivo *qMIPS / qmips.compiler.Anasint.g*

```
header{
	package qmips.compiler;
	import qmips.devices.memory.IMemory;
	import qmips.others.LogicVector;
	import java.util.Map;
	import java.util.Map.Entry;
	import java.util.HashMap;
	import java.util.TreeMap;
	import java.util.Set;
	import java.util.Vector;
}

class Anasint extends Parser;

options{
  k = 6;
}

{

	private IMemory instrMem;
	private IMemory dataMem;
	private int pc = 0;
	private Map<String, Integer> labelMap;
	private Map<Integer, Object[]> solveLater;
	private Map<Integer, Instruction> instructions;
	private Vector<String> compilationErrors;
```

```
	private String name = "";

	public Anasint(Analex l, IMemory instrMem){
		this(l);
		this.instrMem = instrMem;
		this.labelMap = new HashMap<String, Integer>();
		this.solveLater = new HashMap<Integer, Object[]>();
		this.compilationErrors = new Vector<String>();
		this.instructions = new TreeMap<Integer, Instruction>();
	}

	public Anasint(Analex l, IMemory instrMem, IMemory dataMem){
		this(l, instrMem);
		this.dataMem = dataMem;
	}

	private void setLabel(String lbl){
		if(labelMap.containsKey(lbl))
			throw new RuntimeException("Duplicate label: " + lbl);
		labelMap.put(lbl,pc);
	}

	private void solveLabels(){
		Set<Entry<Integer, Object[]>> entrySet = solveLater.entrySet();
		for(Entry<Integer, Object[]> e : entrySet){
			int opcode = (Integer)e.getValue()[0];
			String label = (String)e.getValue()[1];
			if(!labelMap.containsKey(label))
				//throw new RuntimeException("Label: " + label +  " not declared.");
				compilationErrors.add("Label: " + label +  " not declared.");
				else{
					if(opcode == 0x2 || opcode == 0x3){
						int addr =  (labelMap.get(label)/4) & 0x0000FFFF;
						instrMem.load(new LogicVector((opcode << 26) + addr, 32), e.getKey());
					}else{
					int addr = ((labelMap.get(label) - e.getKey()-4)/4) & 0x0000FFFF;
					int s = (Integer)e.getValue()[2];
					int t = (Integer)e.getValue()[3];
					instrMem.load(new LogicVector((opcode << 26) + (s << 21) + (t << 16) + addr, 32),
e.getKey());
					}
					instructions.get(labelMap.get(label)).setLabel(label);
				}
		}
	}

	@Override
	public void reportError(RecognitionException ex){
	compilationErrors.add(ex.getMessage());
	try{
		recover(ex,_tokenSet_0);
	}catch(Exception e){
		compilationErrors.add(e.getMessage());
	}
	}


}


program returns[CompilationResults res = null;]: (dataDirective)* (textDirective)+  EOF {solveLabels(); res = new
CompilationResults(instructions, compilationErrors);}
		;

dataDirective : {int x; int[] v;}(BYTE x=integer v=value
				| HWORD x=integer v=value
				| WORD x=integer v=value )
				{  for(int i = 0; i < v.length; i++){
					if(dataMem == null) instrMem.load(new LogicVector(v[i], 32), x + i*4);
					else dataMem.load(new LogicVector(v[i], 32), x + i*4);
				   }

				}
				;

value returns[int[] res = null]: {int i;} i=integer {res = new int[]{i};}
					| {int i; int[] v;} i=integer v=value
						{int[] aux = new int[v.length +1];
```

```
                                                                    for(int x = 1; x < v.length; x++) aux[x] = v[x];
                                                                    aux[0] = i;
                                                                    res = v;
                                                                   }
                                                ;

integer returns[int i = 0] : n:NUMBER {i = Integer.parseInt(n.getText());}
             | mn:MDEC {i = Integer.parseInt(mn.getText());}
             | h:HEXADECIMAL {i =  Integer.parseInt(h.getText().substring(2),16);}
                                                              | mh:MHEX {i = Integer.parseInt("-" + mh.getText().substring(3),16);}
                                                              ;

textDirective : {int i;} TEXT i=integer {pc = i;} (instructions)+
                          ;

instructions : ({String lbl;} lbl = label {setLabel(lbl);})? body
                          ;

label returns[String l = ""]: a:STRING DP {l = a.getText();}
         | b:STRING {l = b.getText();}
         ;

body : logicArithmetic
        | immediate
        | load
        | store
        | jump
        | jumpr
        | branch
        | mfhi
        | quantum
        | trap
        ;

logicArithmetic :
                                {int funct,d,s,t;}
                                funct = logicArithmeticName d=iregister C s=iregister C t=iregister
                                {
                                        instrMem.load(new LogicVector((s << 21) + (t << 16) + (d << 11) + funct, 32), pc);
                                        instructions.put(pc, new Instruction(name, new String[]{"R" + d, "R" + s, "R" + t}));
                                        pc = pc + 4;
                                }
                                ;

logicArithmeticName returns[int funct = 0]
                                        : SLL  {funct =  0x00; name = "sll";}
                                        | SRL  {funct =  0x02; name = "srl";}
                                        | SRA  {funct =  0x03; name = "sra";}
                                        | ADD  {funct =  0x20; name = "add";}
                                        | ADDU {funct =  0x21; name = "addu";}
                                        | SUB  {funct =  0x22; name = "sub";}
                                        | SUBU {funct =  0x23; name = "subu";}
                                        | MULT {funct =  0x18; name = "mult";}
                                        | DIV  {funct =  0x1A; name = "div";}
                                        | DIVU {funct =  0x1B; name = "divu";}
                                        | AND  {funct =  0x24; name = "and";}
                                        | OR   {funct =  0x25; name = "or";}
                                        | XOR  {funct =  0x26; name = "xor";}
                                        | NOR  {funct =  0x27; name = "nor";}
                                        | SLT  {funct =  0x2A; name = "slt";}
                                        ;

immediate :
                  {int opcode, s, t, imm;}
                  opcode=immediateName t=iregister C s=iregister C imm=integer
                  {
                         instrMem.load(new LogicVector((opcode << 26) + (s << 21) + (t << 16) + (imm & 0x0000FFFF), 32), pc);
                         instructions.put(pc, new Instruction(name, new String[]{"R" + t, "R" + s, "0x" +
Integer.toHexString(imm)}));
                         pc = pc + 4;
                  }
                  ;

immediateName returns[int opcode = 0]
                  : ADDI  {opcode = 0x8; name = "addi";}
                  | ADDIU {opcode = 0x9; name = "addiu";}
                  | ORI   {opcode = 0xD; name = "ori";}
                  | SLTI  {opcode = 0xA; name = "slti";}
```

```
                    ;

load :
     {int opcode, s, t, imm;}
     opcode=loadName t=iregister C imm=integer OP s=iregister CP
          {
                instrMem.load(new LogicVector((opcode << 26) + (s << 21) + (t << 16) + (imm & 0x0000FFFF), 32), pc);
                instructions.put(pc, new Instruction(name, new String[]{"R" + t, "0x" + Integer.toHexString(imm) + "(R" + s +
")"}));
                pc = pc + 4;
          }
         ;

loadName returns[int opcode = 0]
                  : LW  {opcode = 0x23; name = "lw";}
                  ;

store :
      {int opcode, s, t, imm;}
      opcode=storeName imm=integer OP s=iregister CP C t=iregister
      {
       instrMem.load(new LogicVector((opcode << 26) + (s << 21) + (t << 16) + (imm & 0x0000FFFF), 32), pc);
       instructions.put(pc, new Instruction(name, new String[]{"0x" + Integer.toHexString(imm) + "(R" + s + ")", "R" + t}));
       pc = pc+4;
      }
      ;

storeName returns[int opcode = 0]
                   : SW {opcode = 0x2B; name = "sw";}
                   ;

jump :
      {int opcode, addr;}
      opcode=jumpName addr=integer
      {
       instrMem.load(new LogicVector((opcode << 26) + addr, 32), pc);
       instructions.put(pc, new Instruction(name, new String[]{"0x" + Integer.toHexString(addr)}));
       pc = pc+4;
      }
     |
      {int opcode; String lbl;}
      opcode=jumpName lbl=label
      {
       solveLater.put(pc, new Object[]{opcode, lbl});
       instructions.put(pc, new Instruction(name, new String[]{lbl}));
       pc = pc + 4;
      }
     ;

jumpName returns[int opcode = 0]
         : J   {opcode = 0x2; name = "j";}
         | JAL {opcode = 0x3; name = "jal";}
         ;

jumpr:
       {int s;}
       JR s = iregister
       {
                instrMem.load(new LogicVector((0x1B << 26) + (s << 21),32), pc);
                instructions.put(pc, new Instruction("jr", new String[]{"R" + s}));
                pc = pc + 4;
       }
       ;

branch :
       {int opcode, s, t, addr;}
       opcode = branchName s = iregister C t = iregister  C addr = integer
       {
       instrMem.load(new LogicVector((opcode << 26) + (s << 21) + (t << 16) + addr, 32), pc);
       instructions.put(pc, new Instruction(name, new String[]{"R" + s, "R" + t, "0x" + Integer.toHexString(addr)}));
       pc = pc+4;
       }
       |
       {int opcode, s, t; String lbl;}
       opcode = branchName s = iregister C t = iregister  C lbl = label
       {
       solveLater.put(pc, new Object[]{opcode, lbl, s, t});
       instructions.put(pc, new Instruction(name, new String[]{"R" + s, "R" + t, lbl}));
```

```
        pc = pc + 4;
        }
        ;

branchName returns[int opcode = 0]
           : BEQ {opcode = 0x4; name = "beq";}
           | BNE {opcode = 0x5; name = "bne";}
           ;

mfhi : {int s;}
         MFHI s = iregister
         {
                 instrMem.load(new LogicVector((0x1C << 26) + (s << 16), 32), pc);
                 instructions.put(pc, new Instruction("mfhi", new String[]{"R" + s}));
                 pc = pc+4;
         }
         ;

quantum :
         {int target, control, func;}
         func=quantumName target=qregister C control=qregister
         {
                 instrMem.load(new LogicVector((0x0C << 26) + (target << 16) + (control << 11) + func, 32), pc);
                 instructions.put(pc, new Instruction(name, new String[]{"Q" + target, "Q" + control}));
                 pc = pc+4;
         }
         |
         {int target, control, arg;}
         QPHS target=qregister C control=qregister C arg=iregister
         {
                 instrMem.load(new LogicVector((0x0C << 26) + (arg << 21) + (target << 16) + (control << 11) + 0x10, 32), pc);
                 instructions.put(pc, new Instruction("qphs", new String[]{"Q" + target, "Q" + control, "R" + arg}));
                 pc = pc+4;
         }
         |
         {int target, control, arg;}
         QNPH target=qregister C control=qregister C arg=iregister
         {
                 instrMem.load(new LogicVector((0x0C << 26) + (arg << 21) + (target << 16) + (control << 11) + 0x11, 32), pc);
                 instructions.put(pc, new Instruction("qnph", new String[]{"Q" + target, "Q" + control, "R" + arg}));
                 pc = pc+4;
         }
         |
         {int target, reg, arg;}
         QMEA target=qregister C reg = iregister C arg=integer
         {
                 instrMem.load(new LogicVector((0x0F << 26) + (arg << 21) + (target << 16) + (reg << 11) + 0x1A, 32), pc);
                 instructions.put(pc, new Instruction("qmea", new String[]{"Q" + target, "R" + reg, "0x" +
Integer.toHexString(arg)}));
                 pc = pc+4;
         }
         |
         {int reg;}
         QRST reg=iregister
         {
                 instrMem.load(new LogicVector((0x0C << 26) + (reg << 21) + 0x1B, 32), pc);
                 instructions.put(pc, new Instruction("qrst", new String[]{"R" + reg}));
                 pc = pc+4;
         }
         |
         {int reg;}
         QCNT reg=iregister
         {
                 instrMem.load(new LogicVector((0x0C << 26) + (reg << 21) + 0x1C, 32), pc);
                 instructions.put(pc, new Instruction("qcnt", new String[]{"R" + reg}));
                 pc = pc+4;
         }
         |
         {int reg;}
         QOFF reg=iregister
         {
                 instrMem.load(new LogicVector((0x0C << 26) + (reg << 21) + 0x1D, 32), pc);
                 instructions.put(pc, new Instruction("qoff", new String[]{"R" + reg}));
                 pc = pc+4;
         }
         ;

quantumName returns[int func = -1]
```

```
          : QHAD {func = 0x0; name = "qhad";}
          | QX {func = 0x1; name = "qx";}
          | QY {func = 0x2; name = "qy";}
          | QZ {func = 0x3; name = "qz";}
          ;


trap : {int imm;}
         TRAP imm = integer
         {
                  instrMem.load(new LogicVector((0x1A << 26) + imm, 32), pc);
                  instructions.put(pc, new Instruction("trap", new String[]{"0x" + Integer.toHexString(imm)}));
                  pc = pc+4;
         }
         ;


iregister returns[int r = 0]: t:R_REGISTER {r = Integer.parseInt(t.getText().substring(1));}
                  ;

qregister returns[int r = 0]: t:Q_REGISTER {r = Integer.parseInt(t.getText().substring(1));}
     ;
```

Archivo *qMIPS / qmips.devices.Clock.java*

```
/**
 * Representa el reloj principal del sistema. No es hijo de la clase Device
 * porque es un dispositivo especial, que debe ser despertado en el momento
 * oportuno y sin ninguna excitacion externa. De esto se encargan las clases
 * Synchronization.
 *
 * @author Jaime Coello de Portugal
 *
 */
public class Clock implements Runnable {

        private Bus clk;
        private LogicVector lv;
        private int cycleCount = 0, semicycle = 0, remainingCycles = 0;
        private ClockFrame disp;

        public Clock(Bus clk) {
                 this.clk = clk;
                 this.disp = new ClockFrame();
                 lv = new LogicVector(1);
        }

        /**
         *
         * @return el numero de semiciclos transcurridos desde el ultimo reset.
         */
        public int getCycleCount() {
                 return cycleCount;
        }

        /**
         *
         * @param cycleCount
         *            el nuevo numero de semiciclos al que se quiera llevar el
         *            reloj.
         */
        public void setCycleCount(int cycleCount) {
                 this.cycleCount = cycleCount;
                 disp.cycleNumberLabel.setText(cycleCount + "");
                 disp.setMHzTime(cycleCount);
                 if(cycleCount == 0)
                          disp.setTime(0);
        }

        /**
         * Este metodo se ejecuta cada vez que el reloj es despertado y se encarga
         * de dar el flanco correspondiente en el bus clk.
         *
         */
        public void refreshOutput() {
```

```
		if (semicycle == 0) {
			lv.set(0, false);
			clk.write(lv);
			semicycle++;
		} else {
			lv.set(0, true);
			clk.write(lv);
			semicycle--;
			cycleCount++;
			disp.getCycleNumberLabel().setText(cycleCount + "");
			disp.setMHzTime(cycleCount);
		}
	}

	public boolean endCondition() {
		return true;
	}

	/**
	 *
	 * Tarea principal del hilo reloj. Mientras se le permita avanzar llama a ta
	 * tarea que da el flanco de reloj y se duerme a la espera de que la
	 * sincronizacion le despierte.
	 *
	 */
	@Override
	public synchronized void run() {
		while (endCondition()) {
			try {
				while (remainingCycles == 0) {
					wait();
				}
				long tm = System.currentTimeMillis();
				refreshOutput();
				SyncShortcut.sync.taskEnded();
				SyncShortcut.sync.clockLockWait();
				disp.setTime(disp.getTime() + System.currentTimeMillis() - tm);
				remainingCycles--;
			} catch (InterruptedException e) {
				remainingCycles = 0;
				return;
			}
		}
		SyncShortcut.sync.terminate();
	}

	/**
	 * Llamar a este metodo para iniciar el funcionamiento del reloj y por tanto
	 * del todo el simulador.
	 *
	 * @return La referencia al hilo reloj.
	 */
	public Thread startRunning() {
		Thread t = new Thread(this);
		t.start();
		return t;
	}

	/**
	 * Permite al reloj correr el numero de ciclos indicado.
	 *
	 * @param cycleNum
	 *            Numero de ciclos que se quiere que el reloj ejecute
	 *            automaticamente.
	 *
	 */
	public synchronized void runCycles(int cycleNum) {
		remainingCycles = cycleNum * 2;
		notifyAll();
	}

	/**
	 *
	 * Interfaz basica del reloj. Muestra simplemente un numero indicando el
	 * ciclo de reloj actual.
	 *
	 * @author Jaime Coello de Portugal
	 *
```

```
 */
class ClockFrame extends JPanel {
	public ClockFrame() {

		setLayout(new GridLayout(3, 1));
		JPanel panelUp = new JPanel(new BorderLayout());
		JPanel panelCenter = new JPanel(new BorderLayout());
		JPanel panelDown = new JPanel(new BorderLayout());
		add(panelUp);
		add(panelCenter);
		add(panelDown);

		JLabel lblNewLabel = new JLabel("Cycle number");
		lblNewLabel.setHorizontalAlignment(SwingConstants.CENTER);
		panelUp.add(lblNewLabel, BorderLayout.NORTH);

		cycleNumberLabel = new JLabel("0");
		cycleNumberLabel.setFont(new Font("Tahoma", Font.PLAIN, 22));
		cycleNumberLabel.setHorizontalAlignment(SwingConstants.CENTER);
		panelUp.add(cycleNumberLabel, BorderLayout.CENTER);

		JLabel lblNewLabel2 = new JLabel("Time at 25MHz");
		lblNewLabel2.setHorizontalAlignment(SwingConstants.CENTER);
		panelCenter.add(lblNewLabel2, BorderLayout.NORTH);

		timeMhzLabel = new JLabel("0");
		timeMhzLabel.setFont(new Font("Tahoma", Font.PLAIN, 14));
		timeMhzLabel.setHorizontalAlignment(SwingConstants.CENTER);
		panelCenter.add(timeMhzLabel, BorderLayout.CENTER);

		JLabel lblTime = new JLabel("Simulation time");
		lblTime.setHorizontalAlignment(SwingConstants.CENTER);
		panelDown.add(lblTime, BorderLayout.NORTH);

		timeLabel = new JLabel("0");
		timeLabel.setFont(new Font("Tahoma", Font.PLAIN, 14));
		timeLabel.setHorizontalAlignment(SwingConstants.CENTER);
		panelDown.add(timeLabel, BorderLayout.CENTER);

		setVisible(true);
		setPreferredSize(new Dimension(150, 200));
	}

	private static final long serialVersionUID = 9048677220479688042L;
	private JLabel cycleNumberLabel;
	private JLabel timeLabel;
	private JLabel timeMhzLabel;
	private long time;

	public JLabel getCycleNumberLabel() {
		return cycleNumberLabel;
	}

	public JLabel getTimeLabel() {
		return timeLabel;
	}

	public long getTime(){
		return time;
	}

	public void setTime(long time){
		this.time = time;
		timeLabel.setText(String.valueOf(time) + " ms");
	}

	public void setMHzTime(int cycleCount){
		timeMhzLabel.setText(cycleCount / 25 + " us");
	}
}

public JPanel getDisplay() {
	return disp;
}

}
```

Archivo *qMIPS / qmips.devices.control.ControlUnit*

```
public interface ControlUnit {

	public int checkTrap();

	public void releaseTrap();

	public boolean isIF();

}
```

Archivo *qMIPS / qmips.devices.control.ControlUnitDisplay*

```
/**
 *
 * Interfaz grafica para unidades de control generica.
 * Tan solo muestra el estado actual en siglas y una pequeña
 * descrpcion.
 *
 * @author Jaime Coello de Portugal
 *
 */
public class ControlUnitDisplay extends JPanel implements IControlUnitDisplay{

	private static final long serialVersionUID = -7223332015889309434L;
	private JLabel lblState;
	private JLabel lblInstructionFetch;
	public ControlUnitDisplay() {
		setLayout(new BorderLayout(0, 0));

		lblState = new JLabel("IF");
		lblState.setFont(new Font("Tahoma", Font.BOLD, 27));
		lblState.setHorizontalAlignment(SwingConstants.CENTER);
		add(lblState);

		lblInstructionFetch = new JLabel("Instruction fetch");
		lblInstructionFetch.setHorizontalAlignment(SwingConstants.CENTER);
		add(lblInstructionFetch, BorderLayout.SOUTH);

		setSize(180,100);
	}

	public JLabel getLblState() {
		return lblState;
	}
	public JLabel getLblDescription() {
		return lblInstructionFetch;
	}

	@Override
	public void setState(String state) {
		getLblState().setText(state);
	}

	@Override
	public void setDescription(String description) {
		getLblDescription().setText(description);
	}

	@Override
	public void setState(String state, String description) {
		getLblState().setText(state);
		getLblDescription().setText(description);
	}
}
```

Archivo *qMIPS / qmips.devices.control.IControlUnitDisplay*

```
/**
 *
 * Interfaz comun para las interfaces visuales de las unidades
 * de control.
 *
```

```
 * @author Jaime Coello de Portugal
 *
 */
public interface IControlUnitDisplay {

	void setState(String state);

	void setDescription(String description);

	void setState(String state, String description);

}
```

Archivo *qMIPS / qmips.devices.control.QuantumMIPSControlUnit*

```
/**
 *
 * Unidad de control del MIPS cuantico.
 * Es una maquina de estados que evoluciona con el reloj
 * segun las instrucciones que se reciban.
 *
 * @author Jaime Coello de Portugal
 *
 */
public class QuantumMIPSControlUnit extends Device implements ControlUnit {

	Bus clk, rst, opcode;
	Bus machineNotify;
	Bus pcWriteCond, pcWrite, iOrD, memRead, memWrite, memToReg, irWrite,
			pcSource, aluOp, aluSrcB, aluSrcA, regWrite, regDst, solPCWrite,
			aluControl, aluHighWrite, target, qExe, aluOverf;
	State ife, id, mac, mar, maw, mrc, rew, exe, imm, rc, bc, jc, jal, jrf, jrc, mfhi, qt, qex, qmea, trap;
	State current, next;
	IControlUnitDisplay disp;
	int trapNum = -1;
	boolean isIf = false;

	public QuantumMIPSControlUnit(Bus pcWriteCond, Bus pcWrite, Bus iOrD,
			Bus memRead, Bus memWrite, Bus memToReg, Bus irWrite, Bus pcSource,
			Bus aluOp, Bus aluSrcB, Bus aluSrcA, Bus regWrite, Bus regDst,
			Bus solPCWrite, Bus aluControl, Bus aluHighWrite, Bus target, Bus qExe, Bus aluOverf, Bus opcode, Bus clk,
Bus rst) {
		this.pcWriteCond = pcWriteCond;
		this.pcWrite = pcWrite;
		this.iOrD = iOrD;
		this.memRead = memRead;
		this.memWrite = memWrite;
		this.memToReg = memToReg;
		this.irWrite = irWrite;
		this.pcSource = pcSource;
		this.aluOp = aluOp;
		this.aluSrcB = aluSrcB;
		this.aluSrcA = aluSrcA;
		this.regWrite = regWrite;
		this.regDst = regDst;
		this.solPCWrite = solPCWrite;
		this.aluControl = aluControl;
		this.aluHighWrite = aluHighWrite;
		this.target = target;
		this.qExe = qExe;
		this.aluOverf = aluOverf;
		this.clk = clk;
		this.rst = rst;
		this.opcode = opcode;
		this.machineNotify = new Bus(1);
		disp = new SchematicQMIPSControlUnitDisplay();
		this.current = ife;
		defineStates();
		defineBehavior();
	}

	@Override
	protected void defineBehavior() {

		behavior(new Bus[] { clk }, new Behavior() {
```

```
                @Override
                public void task() {
                         if (clk.read().get(0)) {
                                 if(next == null){
                                         current = ife;
                                         current.setOutput();
                                 }else{
                                         current = next;
                                 }
                                 current.setTransition();
                                 machineNotify.write((machineNotify.read().toInteger() + 1) % 2, 1);
                         }
                }

        });

        behavior(new Bus[]{ rst }, new Behavior() {

                @Override
                public void task() {
                        if (rst.read().get(0)) {
                                 current = ife;
                                 disp.setState("IF", "Instruction fetch");
                                 trapNum = -1;
                                 current.setOutput();
                         }
                }
        });

        behavior(new Bus[] { opcode, machineNotify }, new Behavior() {

                @Override
                public void task() {
                        current.setTransition();
                        current.setOutput();
                }

        });

        behavior(new Bus[]{aluOverf}, new Behavior(){

                @Override
                public void task() {
                        if(aluOverf.read().get(0)){
                                 trapNum = -2;
                                 next = trap;
                         }
                }

        });
}

/**
 *
 * Aqui se definen los estados por los que puede pasar la unidad de
 * control. De cada estado se debe describir que salida genera en los
 * buses y a que estado evoluciona.
 *
 */
public void defineStates() {
        ife = new State() {

                @Override
                public void setOutput() {
                        memRead.write(1,1);
                        aluSrcA.write(0,1);
                        iOrD.write(0,1);
                        irWrite.write(1,1);
                        aluSrcB.write(1,2);
                        aluOp.write(0,2);
                        pcWrite.write(1,1);
                        pcSource.write(0,2);

                        regWrite.write(0,1);
                        pcWriteCond.write(0, 1);
                        memWrite.write(0, 1);
                        qExe.write(0, 1);
                        target.write(0, 1);
```

```
                disp.setState("IF", "Instruction fetch");
                isIf = true;
        }

        @Override
        public void setTransition() {
                next =  id;
        }

};

id = new State(){

        @Override
        public void setOutput() {
                aluSrcA.write(0, 1);
                aluSrcB.write(3, 2);
                aluOp.write(0, 2);

                memRead.write(0,1);
                irWrite.write(0,1);
                pcWrite.write(0,1);
                disp.setState("ID", "Instruction decode");
                isIf = false;
        }

        @Override
        public void setTransition() {
                switch(opcode.read().toInteger()){
                case 0x23: //LW
                case 0x2B: //SW
                        next = mac;
                        break;
                case 0: //R-Type
                        next = exe;
                        break;
                case 0x5: //BNE
                case 0x4: //BEQ
                        next = bc;
                        break;
                case 0x2: //J
                        next = jc;
                        break;
                case 0x3: //JAL
                        next = jal;
                        break;
                case 0x8: //ADDI
                        next = imm;
                        break;
                case 0xC: //Q-Type
                case 0xF: //Q-Meas
                        next = qt;
                        break;
                case 0x1A: //TRAP
                        next = trap;
                        break;
                case 0x1B: //JR
                        next = jrf;
                        break;
                case 0x1C: //MFHI
                        next = mfhi;
                        break;
                }
        }

};

mac = new State(){

        @Override
        public void setOutput() {
                aluSrcA.write(1, 1);
                aluSrcB.write(2, 2);
                aluOp.write(0, 2);
                disp.setState("MAC", "Memory address calculation");
        }

        @Override
```

```
		public void setTransition() {
			int intOp = opcode.read().toInteger();
			if(intOp == 0x23){
				next = mar;
			}else{
				next = maw;
			}
		}

};

mar = new State(){

		@Override
		public void setOutput() {
			memRead.write(1,1);
			iOrD.write(1, 1);
			disp.setState("MAR", "Memory read");
		}

		@Override
		public void setTransition() {
			next = mrc;
		}

};

maw = new State(){

		@Override
		public void setOutput() {
			memWrite.write(1,1);
			iOrD.write(1, 1);
			disp.setState("MAW", "Memory write");
		}

		@Override
		public void setTransition() {
			next = ife;
		}

};

mrc = new State(){

		@Override
		public void setOutput() {
			regDst.write(0, 2);
			regWrite.write(1, 1);
			memToReg.write(1, 3);

			memRead.write(0, 1);
			disp.setState("MRC", "Memory to register");
		}

		@Override
		public void setTransition() {
			next = ife;
		}

};

rew = new State(){

		@Override
		public void setOutput() {
			regDst.write(0, 2);
			regWrite.write(1, 1);
			memToReg.write(0, 3);

			memRead.write(0, 1);
			disp.setState("REW", "Register write");
		}

		@Override
		public void setTransition() {
			next = ife;
		}
```

```
};

exe = new State(){

	@Override
	public void setOutput() {
		aluSrcA.write(1, 1);
		aluSrcB.write(0, 2);
		aluOp.write(2, 2);
		aluHighWrite.write(1,1);
		disp.setState("EXE", "Execution");
	}

	@Override
	public void setTransition() {
		next = rc;
	}

};

imm = new State(){

	@Override
	public void setOutput() {
		aluSrcA.write(1, 1);
		aluSrcB.write(2, 2);
		aluOp.write(0, 1);
		disp.setState("IMM", "Immediate execution");
	}

	@Override
	public void setTransition() {
		next = rew;
	}

};

rc = new State(){

	@Override
	public void setOutput() {
		regDst.write(1, 2);
		regWrite.write(1, 1);
		memToReg.write(0, 3);
		aluHighWrite.write(0,1);
		disp.setState("RC", "Register write");
	}

	@Override
	public void setTransition() {
		next = ife;
	}

};

bc = new State(){

	@Override
	public void setOutput() {
		aluSrcA.write(1, 1);
		aluSrcB.write(0, 2);
		aluOp.write(1, 2);
		pcWriteCond.write(1, 1);
		pcSource.write(1, 2);
		disp.setState("BC", "Branch completion");
	}

	@Override
	public void setTransition() {
		next = ife;
	}

};

jc = new State(){

	@Override
```

```
            public void setOutput() {
                    pcWrite.write(1, 1);
                    pcSource.write(2, 2);
                    disp.setState("JC", "Jump completion");
            }

            @Override
            public void setTransition() {
                    next = ife;
            }

    };

    jal = new State(){

            @Override
            public void setOutput() {
                    pcWrite.write(1, 1);
                    pcSource.write(2, 2);
                    regWrite.write(1, 1);
                    memToReg.write(4, 3);
                    regDst.write(2, 2);

                    disp.setState("JAL", "Jump to subroutine");
            }

            @Override
            public void setTransition() {
                    next = ife;
            }

    };

    jrf = new State(){

            @Override
            public void setOutput() {
                    aluSrcA.write(1, 1);
                    aluSrcB.write(0, 2);
                    aluOp.write(0, 2);
                    disp.setState("JRF", "Jump to register forwarding");
            }

            @Override
            public void setTransition() {
                    next = jrc;
            }

    };

    jrc = new State(){

            @Override
            public void setOutput() {
                    pcWrite.write(1, 1);
                    pcSource.write(0, 2);
                    disp.setState("JRC", "Jump to register completion");
            }

            @Override
            public void setTransition() {
                    next = ife;
            }

    };

    mfhi = new State(){

            @Override
            public void setOutput() {
                    regDst.write(0, 2);
                    memToReg.write(2, 3);
                    regWrite.write(1, 1);

                    disp.setState("MFHI", "Move from high");
            }

            @Override
```

```
		public void setTransition() {
			next = ife;
		}

	};

	qt = new State(){

		@Override
		public void setOutput() {
			target.write(1, 1);
			disp.setState("QT", "Quantum target");
		}

		@Override
		public void setTransition() {
			next = qex;
		}

	};

	qex = new State(){

		@Override
		public void setOutput() {
			qExe.write(1, 1);

			target.write(0, 1);
			disp.setState("QEX", "Quantum execution");
		}

		@Override
		public void setTransition() {
			switch(opcode.read().toInteger()){
			case 0xC:
				next = ife;
				break;
			case 0xF:
				next = qmea;
				break;
			}
		}

	};

	qmea = new State(){

		@Override
		public void setOutput() {
			memToReg.write(3, 3);
			regWrite.write(1, 1);
			regDst.write(1, 2);

			qExe.write(0, 1);
			disp.setState("QMEA", "Quantum measuremente write");
		}

		@Override
		public void setTransition() {
				next = ife;
		}

	};

	trap = new State(){

		@Override
		public void setOutput() {
			trapNum = 0;
			disp.setState("TRAP", "Exception " + trapNum);
		}

		@Override
		public void setTransition() {
			next = ife;
		}

	};
```

```
	}

	interface State {

		void setOutput();

		void setTransition();

	}

	@Override
	public JPanel display(){
		return (JPanel)disp;
	}


	/**
	 *
	 * Metodo ofrecido por la unidad de control que la interfaz
	 * de usuario utilizara para saber si se ha producido una
	 * excepcion.
	 *
	 */
	@Override
	public int checkTrap() {
		return trapNum;
	}

	@Override
	public void releaseTrap() {
		trapNum = -1;
	}

	@Override
	public boolean isIF() {
		return isIf;
	}

}
```

Archivo *qMIPS / qmips.devices.control.SchematicQMIPSControlUnitDisplay*

```
public class SchematicQMIPSControlUnitDisplay extends JPanel implements IControlUnitDisplay{

	private static final long serialVersionUID = 1874419105761208518L;

	private Image ife, id, qt, qex, qmea, mac, mar, maw, mrc, imm, rew, exe, rc, jrf, jrc, bc, jc, jal, mfhi;
	private Image current;
	private JPanel imgPanel;
	private JLabel lblDescription;

	public SchematicQMIPSControlUnitDisplay() {
		setLayout(new BorderLayout(0, 0));

		lblDescription = new JLabel("IF: Instruction fetch");
		add(lblDescription, BorderLayout.SOUTH);

		imgPanel = new ImageViewer();
		imgPanel.setBackground(Color.WHITE);
		add(imgPanel, BorderLayout.CENTER);

		ife = new
ImageIcon(SchematicQMIPSControlUnitDisplay.class.getResource("/qmips/devices/control/qMIPSControlImgs/fase_if.png")).getImage();
		id = new
ImageIcon(SchematicQMIPSControlUnitDisplay.class.getResource("/qmips/devices/control/qMIPSControlImgs/fase_id.png")).getImage();
		qt = new
ImageIcon(SchematicQMIPSControlUnitDisplay.class.getResource("/qmips/devices/control/qMIPSControlImgs/fase_qt.png")).getImage();
		qex = new
ImageIcon(SchematicQMIPSControlUnitDisplay.class.getResource("/qmips/devices/control/qMIPSControlImgs/fase_qex.png")).getImage();
		qmea = new
ImageIcon(SchematicQMIPSControlUnitDisplay.class.getResource("/qmips/devices/control/qMIPSControlImgs/fase_qmea.png")).getImage();
		mac = new
ImageIcon(SchematicQMIPSControlUnitDisplay.class.getResource("/qmips/devices/control/qMIPSControlImgs/fase_mac.png")).getImage();
		mar = new
ImageIcon(SchematicQMIPSControlUnitDisplay.class.getResource("/qmips/devices/control/qMIPSControlImgs/fase_mar.png")).getImage();
```

```
                maw = new
ImageIcon(SchematicQMIPSControlUnitDisplay.class.getResource("/qmips/devices/control/qMIPSControlImgs/fase_maw.png")).getImage();
                mrc = new
ImageIcon(SchematicQMIPSControlUnitDisplay.class.getResource("/qmips/devices/control/qMIPSControlImgs/fase_mrc.png")).getImage();
                imm = new
ImageIcon(SchematicQMIPSControlUnitDisplay.class.getResource("/qmips/devices/control/qMIPSControlImgs/fase_imm.png")).getImage();
                rew = new
ImageIcon(SchematicQMIPSControlUnitDisplay.class.getResource("/qmips/devices/control/qMIPSControlImgs/fase_rew.png")).getImage();
                exe = new
ImageIcon(SchematicQMIPSControlUnitDisplay.class.getResource("/qmips/devices/control/qMIPSControlImgs/fase_exe.png")).getImage();
                rc = new
ImageIcon(SchematicQMIPSControlUnitDisplay.class.getResource("/qmips/devices/control/qMIPSControlImgs/fase_rc.png")).getImage();
                jrf = new
ImageIcon(SchematicQMIPSControlUnitDisplay.class.getResource("/qmips/devices/control/qMIPSControlImgs/fase_jrf.png")).getImage();
                jrc = new
ImageIcon(SchematicQMIPSControlUnitDisplay.class.getResource("/qmips/devices/control/qMIPSControlImgs/fase_jrc.png")).getImage();
                bc = new
ImageIcon(SchematicQMIPSControlUnitDisplay.class.getResource("/qmips/devices/control/qMIPSControlImgs/fase_bc.png")).getImage();
                jc = new
ImageIcon(SchematicQMIPSControlUnitDisplay.class.getResource("/qmips/devices/control/qMIPSControlImgs/fase_jc.png")).getImage();
                jal = new
ImageIcon(SchematicQMIPSControlUnitDisplay.class.getResource("/qmips/devices/control/qMIPSControlImgs/fase_jal.png")).getImage();
                mfhi = new
ImageIcon(SchematicQMIPSControlUnitDisplay.class.getResource("/qmips/devices/control/qMIPSControlImgs/fase_mfhi.png")).getImage();

                this.setSize(800, 600);
                this.setPreferredSize(new Dimension(500, 300));
        }

        @Override
        public void setState(String state) {
                if(state.equals("IF")){
                        current = ife;
                }else if(state.equals("ID")){
                        current = id;
                }else if(state.equals("QT")){
                        current = qt;
                }else if(state.equals("QEX")){
                        current = qex;
                }else if(state.equals("QMEA")){
                        current = qmea;
                }else if(state.equals("MAC")){
                        current = mac;
                }else if(state.equals("MAR")){
                        current = mar;
                }else if(state.equals("MAW")){
                        current = maw;
                }else if(state.equals("MRC")){
                        current = mrc;
                }else if(state.equals("IMM")){
                        current = imm;
                }else if(state.equals("REW")){
                        current = rew;
                }else if(state.equals("EXE")){
                        current = exe;
                }else if(state.equals("RC")){
                        current = rc;
                }else if(state.equals("JRF")){
                        current = jrf;
                }else if(state.equals("JRC")){
                        current = jrc;
                }else if(state.equals("BC")){
                        current = bc;
                }else if(state.equals("JC")){
                        current = jc;
                }else if(state.equals("JAL")){
                        current = jal;
                }else if(state.equals("MFHI")){
                        current = mfhi;
                }
                imgPanel.repaint();
        }

        @Override
        public void setDescription(String description) {
                lblDescription.setText(description);
        }
```

```
	@Override
	public void setState(String state, String description) {
		setState(state);
		lblDescription.setText(state + ": " + description);
	}

	class ImageViewer extends JPanel{

		private static final long serialVersionUID = -1982314683516932644L;

		public void paint(Graphics g){
			g.drawImage(current, 0, 0, this.getWidth(), this.getHeight(), null);
		}

	}


}
```

Archivo *qMIPS / qmips.devices.intALU.ALUControl*

```
/**
 *
 * Unidad de control de la ALU tal y como se define
 * en el libro de Hennessy Patterson.
 *
 * @author Jaime Coello de Portugal
 *
 */
public class ALUControl extends Device{

	private Bus func, aluOp, aluControl;

	public ALUControl(Bus func, Bus aluOp, Bus aluControl) {
		this.func = func;
		this.aluOp = aluOp;
		this.aluControl = aluControl;
		defineBehavior();
	}

	@Override
	protected void defineBehavior() {

		behavior(new Bus[]{func, aluOp}, new Behavior() {

			@Override
			public void task() {
				switch(aluOp.read().toInteger()){
				case 0:
					aluControl.write(new LogicVector(IntALU.ADD, 4));
					break;
				case 1:
					aluControl.write(new LogicVector(IntALU.SUB, 4));
					break;
				default:
					switch(func.read().toInteger()){
					case 0x00: //SLL
						aluControl.write(12, 4);
						break;
					case 0x02: //SRL
						aluControl.write(13, 4);
						break;
					case 0x03: //SRA
						aluControl.write(14, 4);
						break;
					case 0x20://ADD
						aluControl.write(new LogicVector(2, 4));
						break;
					case 0x21://ADDU
						aluControl.write(new LogicVector(3, 4));
						break;
					case 0x22://SUB
						aluControl.write(new LogicVector(6, 4));
						break;
					case 0x23://SUBU
```

```
                                        aluControl.write(new LogicVector(7, 4));
                                        break;
                                case 0x24://AND
                                        aluControl.write(new LogicVector(0, 4));
                                        break;
                                case 0x25://OR
                                        aluControl.write(new LogicVector(1, 4));
                                        break;
                                case 0x26://XOR
                                        aluControl.write(new LogicVector(10, 4));
                                        break;
                                case 0x27://NOR
                                        aluControl.write(new LogicVector(11, 4));
                                        break;
                                case 0x2A://SLT
                                        aluControl.write(new LogicVector(15, 4));
                                        break;
                                case 0x18://MULT
                                        aluControl.write(new LogicVector(4, 4));
                                        break;
                                case 0x1A://DIV
                                        aluControl.write(new LogicVector(8, 4));
                                        break;
                                case 0x1B://DIVU
                                        aluControl.write(new LogicVector(9, 4));
                                        break;
                                }
                        }
                }
        });

    }

}
```

Archivo *qMIPS / qmips.devices.intALU.IntALU*

```
/**
 *
 * Unidad aritmetico logica entera.
 * Es facil ver las operaciones que realiza en el codigo.
 *
 * @author Jaime Coello de Portugal
 *
 */
public class IntALU extends Device {

    private Bus a, b, flags, op, outputlow, outputhi;
    public final static int
            AND = 0,
            OR = 1,
            ADD = 2,
            ADDU = 3,
            MULT = 4,
            MULTU = 5,
            SUB = 6,
            SUBU = 7,
            DIV = 8,
            DIVU = 9,
            XOR = 10,
            NOR = 11,
            SLL = 12,
            SRL = 13,
            SRA = 14,
            SLT = 15;

    public IntALU(Bus a, Bus b, Bus op, Bus outputlow, Bus outputhi, Bus flags) {
            this.a = a;
            this.b = b;
            this.flags = flags;
            this.op = op;
            this.outputlow = outputlow;
            this.outputhi = outputhi;
            defineBehavior();
    }
```

```
@Override
protected void defineBehavior() {

	behavior(new Bus[] { a, b, op }, new Behavior() {

		@Override
		public void task() {
			int operation = op.read().toInteger();
			LogicVector opa = a.read();
			int iopa = opa.toInteger();
			LogicVector opb = b.read();
			int iopb = opb.toInteger();
			int ires = 0;
			int upres = 0;
			long aux = 0;
			boolean overf = false;
			boolean negat = false;
			switch (operation) {

			case ADD:
				ires = iopa + iopb;
				if ((iopa > 0 && iopb > 0 && ires < 0)
						|| (iopa < 0 && iopb < 0 && ires > 0))
					overf = true;
				if (ires < 0)
					negat = true;
				break;

			case ADDU:
				ires = iopa + iopb;
				if (ires < 0)
					negat = true;
				break;

			case SUB:
				ires = iopa - iopb;
				if ((iopa > 0 && iopb < 0 && ires < 0)
						|| (iopa < 0 && iopb > 0 && ires > 0))
					overf = true;
				if (ires < 0)
					negat = true;
				break;

			case SUBU:
				ires = iopa - iopb;
				break;

			case MULT:
				aux = (long) iopa * (long) iopb;
				ires = (int) aux;
				upres = (int) (aux >> 32);
				if (upres < 0)
					negat = true;
				break;

			case MULTU:
				aux = (long) iopa * (long) iopb;
				ires = (int) aux;
				upres = (int) (aux >> 32);
				if (upres < 0)
					negat = true;
				break;

			case DIV:
				ires = iopa / iopb;
				upres = iopa % iopb;
				if (ires < 0)
					negat = true;
				break;

			case DIVU:
				ires = iopa / iopb;
				upres = iopa % iopb;
				if (ires < 0)
					negat = true;
				break;
```

```
                                case AND:
                                        ires = iopa & iopb;
                                        break;

                                case OR:
                                        ires = iopa | iopb;
                                        break;

                                case XOR:
                                        ires = iopa ^ iopb;
                                        break;

                                case SLL:
                                        ires = iopa << iopb;
                                        break;

                                case SRL:
                                        ires = iopa >>> iopb;
                                        break;

                                case SRA:
                                        ires = iopa >> iopb;
                                        break;

                                case SLT:
                                        ires = iopa < iopb ? 1 : 0;
                                        break;

                                }

                                LogicVector aflags = new LogicVector(3);
                                if (overf)
                                        aflags.set(0);

                                if (negat)
                                        aflags.set(1);

                                if (ires == 0 && upres == 0)
                                        aflags.set(2);

                                flags.write(aflags);
                                outputhi.write(LogicVector.intToLogicVector(upres));
                                outputlow.write(LogicVector.intToLogicVector(ires));
                        }

                });

        }

}
```

Archivo *qMIPS / qmips.devices.memory.AsyncMemory*

```
/**
 *
 * Memoria que responde automaticamente para lecturas
 * pero se escribe en el flanco de subida.
 * Se puede definir su tamaño en numero de parabras de 32 bits.
 *
 * @author Jaime Coello de Portugal
 *
 */
public class AsyncMemory extends Device implements IMemory{

        private LogicVector[] memContents;
        private Bus input, output, addr, rd, wr, clk;

        public AsyncMemory(Bus input, Bus output, Bus addr, Bus rd, Bus wr, Bus clk, int words) {
                this.input = input;
                this.output = output;
                this.rd = rd;
                this.wr = wr;
                this.clk = clk;
                this.addr = addr;
```

```
		memContents = new LogicVector[words];
		for(int i = 0; i < words; i++){
			memContents[i] = new LogicVector(32);
		}
		defineBehavior();
	}

	public void load(LogicVector v, int dir){
		memContents[dir/4] = v;
		int dir2 = addr.read().toInteger()/4;
		if(dir/4 == dir2){
			output.write(memContents[dir/4]);
		}
	}

	@Override
	protected void defineBehavior() {

		behavior(new Bus[]{rd, addr}, new Behavior(){
			@Override
			public void task() {
				if(rd.read().get(0)){
					int dir = addr.read().toInteger()/4;
					if(dir >= 0){
						output.write(memContents[dir]);
					}
				}
			}
		});

		behavior(new Bus[]{clk}, new Behavior(){
			@Override
			public void task() {
				if(wr.read().get(0)){
					int dir = addr.read().toInteger()/4;
					if(dir >= 0){
						memContents[dir] = input.read();
					}
				}
			}
		});

	}

	@Override
	public int size() {
		return memContents.length;
	}


}
```

Archivo *qMIPS / qmips.devices.memory.IMemory*

```
/**
 *
 * Interfaz utilizada para decirle al compilador que
 * puede insertar el codigo compilado con la funcion load(...).
 *
 * @author Jaime Coello de Portugal
 *
 */
public interface IMemory {

	public void load(LogicVector v, int dir);

	public int size();

}
```

Archivo *qMIPS / qmips.devices.memory.Memory*

```
/**
```

```
 *
 * Memoria totalmente sincrona.
 * Lee y escribe en el flanco de subida del reloj.
 *
 * @author Jaime Coello de Portugal
 *
 */
public class Memory extends Device implements IMemory{

	private LogicVector[] memContents;
	private Bus input, output, addr, rd, wr, clk;

	public Memory(Bus input, Bus output, Bus addr, Bus rd, Bus wr, Bus clk, int words) {
		this.input = input;
		this.output = output;
		this.rd = rd;
		this.wr = wr;
		this.clk = clk;
		this.addr = addr;
		memContents = new LogicVector[words];
		for(int i = 0; i < words; i++){
			memContents[i] = new LogicVector(32);
		}
		defineBehavior();
	}

	public void load(LogicVector v, int dir){
		memContents[dir/4] = v;
	}

	@Override
	protected void defineBehavior() {
		behavior(new Bus[]{clk}, new Behavior() {

			@Override
			public void task() {
				if(clk.read().get(0) && wr.read().get(0)){
					memContents[addr.read().toInteger()/4] = input.read();
				}
				if(clk.read().get(0) && rd.read().get(0)){
					int intaddr = addr.read().toInteger()/4;
					if(intaddr >= 0){
						output.write(memContents[intaddr]);
					}
				}
			}

		});
	}

	@Override
	public int size() {
		return memContents.length;
	}

}
```

Archivo *qMIPS / qmips.devices.quantum.QuantumControl*

```
/**
 *
 * Este dispositivo opera sobre el array de qubits.
 * Las operaciones que realiza se pueden ver en el codigo.
 *
 * @author Jaime Coello de Portugal
 *
 */
public class QuantumControl extends Device{

	private Bus param, funct, regValue, qexe, clk, rst, meas;
	private QubitArray32 qarray;
	private Display disp;
	private SortedSet<Integer> controlQubits;
	private int offset;
```

```
	public QuantumControl(Bus funct, Bus param, Bus regValue, Bus meas, Bus qexe, Bus clk, Bus rst, QubitArray32 qarray){
		this.param = param;
		this.funct = funct;
		this.regValue = regValue;
		this.meas = meas;
		this.qexe = qexe;
		this.clk = clk;
		this.rst = rst;
		this.qarray = qarray;
		this.controlQubits = new TreeSet<Integer>();
		this.offset = 0;
		this.disp = new QuantumControlDisplay();
		disp.updateText(qarray.getState().toString(), Integer.toString(offset), controlQubits.toString());
		defineBehavior();
	}

	@Override
	protected void defineBehavior() {

		behavior(new Bus[]{clk}, new Behavior(){

			@Override
			public void task() {
				if(clk.read().get(0)){
					if(qexe.read().get(0)){
						int ifunct = funct.read().toInteger();
						QuantumState state = qarray.getState();
						int[] control;
						Vector<Integer> v = new Vector<Integer>();
						for(Integer i : controlQubits){
							if(i != ((qarray.getSelectedTarget() + offset) % 32))
								v.add(i);
						}
						if(qarray.getSelectedControl() != -1)
							v.add((qarray.getSelectedControl() + offset) % 32);
						control = new int[v.size()];
						for(int i = 0; i < control.length; i++){
							control[i] = v.get(i);
						}
						switch(ifunct){
						case 0x00: //QHAD
							HadamardGate hg = new HadamardGate();
							qarray.setState(hg.operate(state, (qarray.getSelectedTarget() +
offset) % 32, control));
							break;
						case 0x01: //QX
							PauliXGate px = new PauliXGate();
							qarray.setState(px.operate(state, (qarray.getSelectedTarget() +
offset) % 32, control));
							break;
						case 0x02: //QY
							PauliYGate py = new PauliYGate();
							qarray.setState(py.operate(state, (qarray.getSelectedTarget() +
offset) % 32, control));
							break;
						case 0x03: //QZ
							PauliZGate pz = new PauliZGate();
							qarray.setState(pz.operate(state, (qarray.getSelectedTarget() +
offset) % 32, control));
							break;
						case 0x10: //QPHS
							PhaseShiftGate ph = new PhaseShiftGate("(2*pi)/(2^" +
regValue.read().toInteger() + ")");
							qarray.setState(ph.operate(state, (qarray.getSelectedTarget() +
offset) % 32, control));
							break;
						case 0x11: //QNPH
							PhaseShiftGate np = new PhaseShiftGate("(-2*pi)/(2^" +
regValue.read().toInteger() + ")");
							qarray.setState(np.operate(state, (qarray.getSelectedTarget() +
offset) % 32, control));
							break;
						case 0x1A: //QMEA
							Measure m = new Measure();
							qarray.setState(m.operate(state, qarray.getSelectedTarget()));
							meas.write(m.getMeasurementResult() << param.read().toInteger(), 32);
							break;
						case 0x1B: //QRST
```

```
                                                        QuantumState qs = new QuantumState();
                                                        byte[] val = new byte[32];
                                                        LogicVector lv = regValue.read();
                                                        for(int i = 0; i < 32; i++)
                                                                val[i] = lv.get(i) ? (byte)1 : (byte)0;
                                                        qs.add(new Complex(1.0,0.0), new ClassicState(val));
                                                        qarray.setState(qs);
                                                        break;
                                                case 0x1C: //QCNT
                                                        int index = ((0X0000001F & regValue.read().toInteger()) + offset) %
32;
                                                        if(controlQubits.contains(index)){
                                                                controlQubits.remove(index);
                                                        }else{
                                                                controlQubits.add(index);
                                                        }
                                                        break;
                                                case 0x1D: //QOFF
                                                        offset = 0X0000001F & regValue.read().toInteger();
                                                        break;
                                                }
                                                disp.updateText(qarray.getState().toString(), Integer.toString(offset),
controlQubits.toString());
                                        }
                                }
                        }

                });

                behavior(new Bus[]{ rst }, new Behavior() {

                        @Override
                        public void task() {
                                QuantumState qs = new QuantumState();
                                qs.add(new Complex(1.0,0.0), new ClassicState(new byte[32]));
                                qarray.setState(qs);
                                controlQubits.clear();
                                offset = 0;
                                disp.updateText(qarray.getState().toString(), Integer.toString(offset),
controlQubits.toString());
                        }

                });
        }

        public JPanel display(){
                return (JPanel)disp;
        }

        interface Display{
                void updateText(String s, String o, String c);
        }
}
```

Archivo *qMIPS / qmips.devices.quantum.QuantumControlDisplay*

```
/**
 *
 * Interfaz grafica simple para el sistema de informacion
 * cuantica.
 * Muestra simplemente el estado actual del array de qubits
 * en formato texto.
 *
 * @author Jaime Coello de Portugal
 *
 */
public class QuantumControlDisplay extends JPanel implements QuantumControl.Display{
        public QuantumControlDisplay() {
                setLayout(new BorderLayout(0, 0));

                setSize(350, 395);

                JPanel panel = new JPanel();
                add(panel, BorderLayout.CENTER);
                panel.setLayout(new BorderLayout(0, 0));
```

```
		JScrollPane scroll = new JScrollPane();
		panel.add(scroll);

		stateText = new JTextArea();
		scroll.setViewportView(stateText);
		stateText.setEditable(false);

		JPanel panel_1 = new JPanel();
		panel.add(panel_1, BorderLayout.SOUTH);
		panel_1.setLayout(new BoxLayout(panel_1, BoxLayout.Y_AXIS));

		JLabel lblOffset = new JLabel("Offset:");
		lblOffset.setFont(new Font("Tahoma", Font.BOLD, 11));
		lblOffset.setHorizontalAlignment(SwingConstants.CENTER);
		panel_1.add(lblOffset);

		JLabel label = new JLabel("");
		panel_1.add(label);

		JLabel label_2 = new JLabel("");
		panel_1.add(label_2);

		JLabel label_3 = new JLabel("");
		panel_1.add(label_3);

		offsetLbl = new JLabel("0");
		offsetLbl.setAlignmentY(Component.TOP_ALIGNMENT);
		panel_1.add(offsetLbl);

		JLabel lblControlQubits = new JLabel("Control Qubits:");
		lblControlQubits.setFont(new Font("Tahoma", Font.BOLD, 11));
		panel_1.add(lblControlQubits);

		controlLbl = new JLabel("[]");
		panel_1.add(controlLbl);
		setVisible(true);
	}

	private static final long serialVersionUID = -5305181440555107289L;
	private JTextArea stateText;
	private JLabel offsetLbl;
	private JLabel controlLbl;

	@Override
	public void updateText(String s, String o, String c) {
		getStateText().setText(s);
		getOffsetLbl().setText(o);
		getControlLbl().setText(c);
		repaint();
	}

	public JTextArea getStateText() {
		return stateText;
	}
	protected JLabel getOffsetLbl() {
		return offsetLbl;
	}
	protected JLabel getControlLbl() {
		return controlLbl;
	}
}
```

Archivo *qMIPS / qmips.devices.quantum.QubitArray32*

```
/**
 *
 * Contiene un array de 32 qubits sobre el que se realizaran
 * las operaciones cuanticas.
 * El tipo QubitTargetControl define sobre que qubits concretos
 * se actuara y QuantumControl realiza las operaciones correspondientes
 * sobre este array.
 *
 * El simulador de estados cuanticos se vuelve mas lento con el tamaño
```

```
 * de la superposicion del array. Al movernos en el orden de 20 qubits
 * en superposicion el sistema puede colgarse por falta de memoria.
 *
 * La clase QuantumState esta importada del proyecto Qubit101.
 *
 * @author Jaime Coello de Portugal
 *
 */
public class QubitArray32{

	private QuantumState state;
	private int selectedTarget = -1;
	private int selectedControl = - 1;

	public QubitArray32(){
		state = new QuantumState();
		state.add(new Complex(1.0,0.0), new ClassicState(new byte[32]));
	}

	public int getSelectedTarget() {
		return selectedTarget;
	}

	public void setSelectedTarget(int selectedTarget) {
		this.selectedTarget = selectedTarget;
	}

	public int getSelectedControl() {
		return selectedControl;
	}

	public void setSelectedControl(int selectedControl) {
		this.selectedControl = selectedControl;
	}

	public QuantumState getState(){
		return state;
	}

	public void setState(QuantumState state){
		this.state = state;
	}
}
```

Archivo *qMIPS / qmips.devices.quantum.QubitArray32*

```
/**
 *
 * Dispositivo que controla sobre que qubits del array se opera.
 * Si se indica el mismo qubit para control y objetivo se supone
 * que la operacion es no controlada.
 *
 * @author Jaime Coello de Portugal
 *
 */
public class QubitTargetControl extends Device{

	private Bus qubit, control, target, clk;
	private QubitArray32 qarray;

	public QubitTargetControl(Bus qubit, Bus control, Bus target, Bus clk, QubitArray32 qarray){
		this.qubit = qubit;
		this.control = control;
		this.target = target;
		this.clk = clk;
		this.qarray = qarray;
		defineBehavior();
	}

	@Override
	protected void defineBehavior() {
		behavior(new Bus[]{clk}, new Behavior(){
			@Override
			public void task() {
				if(clk.read().get(0) && target.read().get(0)){
					int iq = qubit.read().toInteger();
```

```
                                        int ic = control.read().toInteger();
                                        boolean controlled = iq != ic;
                                        qarray.setSelectedTarget(iq);
                                        qarray.setSelectedControl(controlled? ic : -1);
                                }
                        }
                });
        }

}
```

Archivo *qMIPS / qmips.devices.registerFile.RegisterFile*

```
/**
 *
 * Fichero de 32 registros de 32 bits.
 * Las lecturas son asincronas, el dispositivo responde
 * inmediatamente y se pueden leer dos datos a la vez.
 * Las escrituras se realizan en el flanco de subida.
 *
 * @author Jaime Coello de Portugal
 *
 */
public class RegisterFile extends Device {

        private Bus selA, selB, outputA, outputB, selW, wr, input, rst, clk;
        private LogicVector[] contents;
        private Display disp;

        public RegisterFile(Bus selA, Bus selB, Bus outputA, Bus outputB, Bus selW,
                        Bus wr, Bus input, Bus rst, Bus clk) {
                this.clk = clk;
                this.rst = rst;
                this.input = input;
                this.outputA = outputA;
                this.outputB = outputB;
                this.selA = selA;
                this.selB = selB;
                this.selW = selW;
                this.wr = wr;
                this.contents = new LogicVector[32];
                for (int i = 0; i < 32; i++) {
                        contents[i] = new LogicVector(32);
                }
                disp = new RegisterFileDisplay();
                defineBehavior();
        }

        public void load(int index, LogicVector value) {
                this.contents[index] = value;
        }

        @Override
        protected void defineBehavior() {

                behavior(new Bus[] { clk }, new Behavior() {

                        @Override
                        public void task() {
                                if (clk.read().get(0) && wr.read().get(0)) {
                                        disp.write(selW.read().toInteger(), input.read());
                                        contents[selW.read().toInteger()] = input.read();
                                }
                        }

                });

                behavior(new Bus[] {selA, selB}, new Behavior() {

                        @Override
                        public void task() {
                                int iselA = selA.read().toInteger();
                                int iselB = selB.read().toInteger();
                                disp.setSelectedA(iselA);
                                disp.setSelectedB(iselB);
                                if (iselA == 0) {
                                        outputA.write(new LogicVector(32));
                                } else {
                                        outputA.write(contents[iselA]);
```

```
                                }
                                if (iselB == 0) {
                                        outputB.write(new LogicVector(32));
                                } else {
                                        outputB.write(contents[iselB]);
                                }
                        }

                });

                behavior(new Bus[]{rst}, new Behavior(){

                        @Override
                        public void task() {
                                if(rst.read().get(0)){
                                        contents = new LogicVector[32];
                                        for (int i = 0; i < 32; i++) {
                                                contents[i] = new LogicVector(32);
                                                disp.write(i, new LogicVector(32));
                                        }
                                }
                        }

                });

        }

        @Override
        public JPanel display(){
                return (JPanel) disp;
        }

        public interface Display{

                void setSelectedA(int iselA);
                void setSelectedB(int iselB);

                void write(int reg, LogicVector value);
                void reset();

        }

}
```

Archivo *qMIPS / qmips.devices.registerFile.RegisterFileDisplay*

```
/**
 *
 * Interfaz grafica simple para el fichero de registros.
 *
 * @author Jaime Coello de Portugal
 *
 */
public class RegisterFileDisplay extends JPanel implements RegisterFile.Display {

        private static final long serialVersionUID = -5224894535840510807L;
        private JLabel selRegA;
        private JLabel selRegB;
        private JList<String> list;
        private String[] elems;

        public RegisterFileDisplay() {
                setSize(400, 430);
                setLayout(new BorderLayout(0, 0));

                elems = new String[32];
                for (int i = 0; i < 32; i++)
                        elems[i] = "R" + i + ": " + new LogicVector(32).toString();

                JScrollPane scroll = new JScrollPane();
                list = new JList<String>(elems);
                list.setVisibleRowCount(32);
                list.setBorder(new LineBorder(new Color(0, 0, 0)));
```

```
		list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
		scroll.setViewportView(list);
		add(scroll);

		JPanel panel = new JPanel();
		add(panel, BorderLayout.WEST);

		JPanel panel_1 = new JPanel();
		panel_1.setAlignmentX(Component.RIGHT_ALIGNMENT);
		panel_1.setLayout(new BorderLayout(0, 0));

		JLabel lblSelectedRegisterA = new JLabel("Selected register A");
		lblSelectedRegisterA.setHorizontalAlignment(SwingConstants.CENTER);
		lblSelectedRegisterA.setFont(new Font("Tahoma", Font.BOLD, 14));
		panel_1.add(lblSelectedRegisterA, BorderLayout.NORTH);

		selRegA = new JLabel("0");
		selRegA.setFont(new Font("Tahoma", Font.BOLD, 18));
		selRegA.setHorizontalAlignment(SwingConstants.CENTER);
		panel_1.add(selRegA);
		panel.setLayout(new FormLayout(new ColumnSpec[] {
				FormFactory.LABEL_COMPONENT_GAP_COLSPEC,
				ColumnSpec.decode("center:135px"), }, new RowSpec[] {
				FormFactory.LINE_GAP_ROWSPEC, RowSpec.decode("39px"),
				FormFactory.RELATED_GAP_ROWSPEC, FormFactory.DEFAULT_ROWSPEC, }));
		panel.add(panel_1, "2, 2, left, top");

		JPanel panel_2 = new JPanel();
		panel_2.setLayout(new BorderLayout(0, 0));

		JLabel lblSelectedRegisterB = new JLabel("Selected register B");
		lblSelectedRegisterB.setHorizontalAlignment(SwingConstants.CENTER);
		lblSelectedRegisterB.setFont(new Font("Tahoma", Font.BOLD, 14));
		panel_2.add(lblSelectedRegisterB, BorderLayout.NORTH);

		selRegB = new JLabel("0");
		selRegB.setFont(new Font("Tahoma", Font.BOLD, 18));
		selRegB.setHorizontalAlignment(SwingConstants.CENTER);
		panel_2.add(selRegB);
		panel.add(panel_2, "2, 4, left, top");
	}

	@Override
	public void setSelectedA(int iselA) {
		selRegA.setText(String.valueOf(iselA));
	}

	@Override
	public void setSelectedB(int iselB) {
		selRegB.setText(String.valueOf(iselB));
	}

	@Override
	public void write(int reg, LogicVector value) {
		elems[reg] = "R" + reg + ": " + value.toString();
		list.repaint();
	}

	public JLabel getSelRegA() {
		return selRegA;
	}

	public JLabel getSelRegB() {
		return selRegB;
	}

	public JList<String> getRegList() {
		return list;
	}

	@Override
	public void reset() {
		elems = new String[32];
		for (int i = 0; i < 32; i++)
			elems[i] = "R" + i + ": " + new LogicVector(32).toString();
		repaint();
	}
}
```

Archivo *qMIPS / qmips.devices.simple.Concat*

```
/**
 *
 * Dispositivo que concatena una serie de buses en uno solo.
 *
 * @author Jaime Coello de Portugal
 *
 */
public class Concat extends Device{

	private Bus[] buses;
	private Bus output;

	public Concat(Bus[] buses, Bus output){
		this.buses = buses;
		this.output = output;
		defineBehavior();
	}

	@Override
	protected void defineBehavior() {
		behavior(buses, new Behavior(){

			@Override
			public void task() {
				int size = 0;
				int res = 0;
				for(int i = buses.length - 1; i >= 0; i--){
					res += buses[i].read().toInteger() << size;
					size += buses[i].size();
				}
				output.write(new LogicVector(res, size));
			}

		});
	}

}
```

Archivo *qMIPS / qmips.devices.simple.Logic*

```
/**
 *
 * Puertas logicas genericas.
 * Compatible con cualquier ancho de bus de entrada.
 * Se concreta la puerta logica en el constructor.
 *
 * @author Jaime Coello de Portugal
 *
 */
public class Logic extends Device{

	public static final int AND = 0, OR = 1, NAND = 2, XOR = 3, NOR = 4, XNOR = 5;
	private Bus i1, i2, o;
	private int op;

	public Logic(Bus i1, Bus i2, Bus o, int op) {
		this.i1 = i1;
		this.i2 = i2;
		this.o = o;
		this.op = op;
		defineBehavior();
	}

	@Override
	protected void defineBehavior() {

		behavior(new Bus[]{i1, i2}, new Behavior() {

			@Override
			public void task() {
				int ii1 = i1.read().toInteger();
```

```
                                int ii2 = i2.read().toInteger();
                                int res = 0;
                                switch(op){
                                case AND:
                                        res = ii1 & ii2;
                                        break;
                                case OR:
                                        res = ii1 | ii2;
                                        break;
                                case NAND:
                                        res = ~(ii1 & ii2);
                                        break;
                                case XOR:
                                        res = ii1 ^ ii2;
                                        break;
                                case NOR:
                                        res = ~(ii1 | ii2);
                                        break;
                                case XNOR:
                                        res = ~(ii1 ^ ii2);
                                        break;
                                }
                                o.write(new LogicVector(res, o.size()));
                        }

                });

        }


}
```

Archivo *qMIPS / qmips.devices.simple.Multiplexer*

```
/**
 *
 * Multiplexor de cualquier numero de entradas.
 *
 * @author Jaime Coello de Portugal
 *
 */
public class Multiplexer extends Device{

        private Bus control;
        private Bus inputs[];
        private Bus output;


        public Multiplexer(Bus control, Bus inputs[], Bus output) {
                this.control = control;
                this.inputs = inputs;
                this.output = output;
                defineBehavior();
        }

        private static Bus[] solveWires(Bus control, Bus inputs[]){
                Bus[] res = new Bus[inputs.length + 1];
                System.arraycopy(inputs, 0, res, 0, inputs.length);
                res[inputs.length] = control;
                return res;
        }


        @Override
        protected void defineBehavior() {
                behavior(solveWires(control, inputs), new Behavior() {

                        @Override
                        public void task() {
                                int index = control.read().toInteger();
                                output.write(inputs[index].read());
                        }

                });
```

```
	}

}
```

Archivo *qMIPS / qmips.devices.simple.Multiplexer*

```
/**
 *
 * Dispositivo que desplaza a la izquierda el valor
 * de la entrada la cantidad indicada.
 *
 * @author Jaime Coello de Portugal
 *
 */
public class ShiftLeft extends Device{

	private Bus input, output;
	private int amount;

	public ShiftLeft(Bus input, Bus output, int amount) {
		this.input = input;
		this.output = output;
		this.amount = amount;
		defineBehavior();
	}

	@Override
	protected void defineBehavior() {

		behavior(new Bus[] { input }, new Behavior() {

			@Override
			public void task() {
				output.write(new LogicVector(input.read().toInteger() << amount, 32));
			}
		});

	}


}
```

Archivo *qMIPS / qmips.devices.simple.SignExt*

```
/**
 *
 * Extensor de signo de 16 a 32 bits.
 *
 * @author Jaime Coello de Portugal
 *
 */
public class SignExt extends Device {

	Bus input, output;

	public SignExt(Bus input, Bus output) {
		this.input = input;
		this.output = output;
		defineBehavior();
	}

	@Override
	protected void defineBehavior() {

		behavior(new Bus[] { input }, new Behavior() {

			@Override
			public void task() {
				LogicVector res = new LogicVector(32);
				LogicVector in = input.read();
				boolean sign = in.get(15);
```

```
                                for (int i = 0; i < 32; i++) {
                                        if (i <= 15)
                                                res.set(i, in.get(i));
                                        else
                                                res.set(i, sign);
                                }
                                output.write(res);
                        }
                });
        }

}
```

Archivo *qMIPS / qmips.devices.simple.SynchronousRegister*

```
/**
 *
 * Registro sincrono simple.
 * Se carga un valor con 'en' activo en el flanco de subida.
 *
 * @author Jaime Coello de Portugal
 *
 */
public class SynchronousRegister extends Device {

        Bus input, output, en, clk, rst;
        Display disp;
        LogicVector content;

        public SynchronousRegister(Bus input, Bus output, Bus en, Bus rst, Bus clk) {
                this.input = input;
                this.output = output;
                this.en = en;
                this.rst = rst;
                this.clk = clk;
                content = new LogicVector(output.size());
                disp =  new SynchronousRegisterDisplay();
                defineBehavior();
        }


        @Override
        protected void defineBehavior() {
                behavior(new Bus[] { clk }, new Behavior() {

                        @Override
                        public void task() {
                                 if (clk.read().get(0)) {
                                        if (en.read().get(0)) {
                                                content = input.read();
                                                disp.setContent(input.read());
                                                output.write(input.read());
                                        }
                                }
                        }

                });

                behavior(new Bus[]{ rst }, new Behavior() {

                        @Override
                        public void task() {
                                if (rst.read().get(0)) {
                                        content = new LogicVector(output.size());
                                        output.write(new LogicVector(output.size()));
                                        disp.setContent(new LogicVector(output.size()));
                                }
                        }

                });

        }

        public LogicVector getContent(){
                return content;
```

```
	}

	@Override
	public JPanel display(){
		return (JPanel) disp;
	}

	public interface Display{
		void setContent(LogicVector v);
	}

	/**
	 *
	 * Interfaz grafica simple para los registros.
	 *
	 * @author Jaime Coello de Portugal
	 *
	 */
	class SynchronousRegisterDisplay extends JPanel implements Display{

		private static final long serialVersionUID = -5586274312021370406L;
		JLabel content;

		public SynchronousRegisterDisplay(){
			this.setSize(360, 30);
			content = new JLabel(new LogicVector(32).toString());
			content.setBackground(Color.WHITE);
			add(content);
		}

		@Override
		public void setContent(LogicVector v) {
			content.setText(v.toString());
		}

	}

}
```

Archivo *qMIPS / qmips.devices.Devices*

```
/**
 *
 * Esta clase debe ser padre de todos los dispositivos comunes.
 *
 *
 * @author Jaime Coello de Portugal
 *
 */
public abstract class Device {

	/**
	 *
	 * Define la reaccion que debe tener el dispositivo a los cambios
	 * en ciertos buses.
	 * Se puede llamar mas de una vez si se quieren definir reacciones distintas
	 * a distintos buses.
	 *
	 * @param sensivity Los buses a los que reaccionara el dispositivo. Lo que en
	 * lenguajes de descripcion de hardware se llama "lista de sensibilidad".
	 * @param process El proceso que se ejecutara si cambia alguno de los buses.
	 */
	protected void behavior(Bus[] sensivity, Behavior process){
		for(Bus b : sensivity){
			b.addProcess(process);
		}
	}

	/**
	 *
	 * Metodo para definir el comportamiento del dispositivo. Debe contener llamadas
	 * a behavior(...). Se debe invocar al final del contructor.
	 *
	 */
	protected abstract void defineBehavior();
```

```
	/**
	 * Este metodo debe ser sobreescrito por los dispositivos con interfaz.
	 *
	 * @return La interfaz del dispositivo.
	 */
	public JPanel display(){
		return null;
	}

}
```

Archivo *qMIPS / qmips.others.Behavior*

```
/**
 *
 * Clase abstracta para definir las tareas de cada
 * dispositivo.
 * Avisa a la sincronizacion al terminar la tarea.
 *
 * @author Jaime Coello de Portugal
 *
 */
public abstract class Behavior implements Runnable{

	public void run(){
		try{
			task();
		}catch(Exception e){

		}
		SyncShortcut.sync.taskEnded();
	}

	public abstract void task();

}
```

Archivo *qMIPS / qmips.others.Bus*

```
/**
 *
 * Clase para definir un bus de cualquier tamaño.
 *
 *
 * @author Jaime Coello de Portugal
 *
 */
public class Bus {

	private Bus parentBus = null;
	private Set<Runnable> processes;
	private LogicVector content;
	private boolean trace = false;
	private String name;

	public Bus(int size) {
		processes = new HashSet<Runnable>();
		content = new LogicVector(size);
	}

	public Bus(LogicVector content) {
		processes = new HashSet<Runnable>();
		this.content = content;
	}

	public Bus(int content, int size) {
		this(new LogicVector(content, size));
	}

	public Bus addProcess(Runnable process) {
```

```
                processes.add(process);
                if (parentBus != null)
                        parentBus.addProcess(process);
                return this;
        }

        public Bus getRange(int from, int to) {
                Bus res = new Bus(content.get(from, to));
                res.parentBus = this;
                for (Runnable r : processes)
                        res.addProcess(r);
                return res;
        }

        public LogicVector read() {
                LogicVector res = new LogicVector(this.size());
                for (int i = 0; i < this.size(); i++)
                        res.set(i, content.get(i));
                return res;
        }

        public synchronized void write(int value, int size){
                write(new LogicVector(value, size));
        }

        public synchronized void write(LogicVector in) {
                if (trace)
                        System.out.println("[" + name + "] " + " written: " + in);
                if (!in.equals(content)) {
                        for (int i = 0; i < content.size() && i < in.size(); i++)
                                content.set(i, in.get(i));
                        SyncShortcut.sync.activateProcesses(processes);
                }
        }

        public int size() {
                return content.size();
        }

        public static Bus[] createBusArray(int busSize, int arraySize) {
                Bus[] res = new Bus[arraySize];
                for (int i = 0; i < arraySize; i++)
                        res[i] = new Bus(busSize);
                return res;
        }

        public Bus trace(String name){
                trace = true;
                this.name = name;
                return this;
        }

        public void untrace(){
                trace = false;
        }

}
```

Archivo *qMIPS / qmips.others.LogicVector*

```
public class LogicVector{

        private HBoolean[] array;

        public LogicVector(int size){
                array = new HBoolean[size];
                for(int i = 0; i < size; i++)
                        array[i] = new HBoolean();
        }

        public LogicVector(Boolean[] values){
                HBoolean[] res = new HBoolean[values.length];
                for(int i = 0; i < values.length; i++){
```

```
                res[i] = new HBoolean();
                res[i].value = values[i];
        }
}

public LogicVector(int number, int size){
        this(size);
        LogicVector lv = LogicVector.intToLogicVector(number);
        for(int i = 0; i < size; i++)
                array[i].value = lv.get(i);
}

public LogicVector(String bin){
        array = new HBoolean[bin.length()];
        char[] chr = bin.toCharArray();
        for(int i = 0; i < array.length; i++){
                HBoolean aux = new HBoolean();
                if(chr[i] == '1')
                        aux.value = true;
                else if(chr[i] != '0')
                        throw new IllegalArgumentException();
                array[i] = aux;
        }
}

private LogicVector(HBoolean[] values){
        array = values;
}

public Boolean get(int i){
        return array[i].value;
}

public LogicVector get(int from, int to){
        HBoolean[] res = new HBoolean[to - from];
        System.arraycopy(array, from, res, 0, to - from);
        return new LogicVector(res);
}

public void set(int i){
        array[i].value = true;
}

public void set(int i, boolean value){
        array[i].value = value;
}

public void set(int from, int to){
        for(int i = from; i < to; i++)
                array[i].value = true;
}

public int size(){
        return array.length;
}

public int toInteger(){
        int bitInteger = 0;
        int size = (size() < 32) ? size(): 32;
    for(int i = 0 ; i < size; i++)
        if(get(i))
           bitInteger |= (1 << i);
    return bitInteger;
}

public static LogicVector intToLogicVector(int value)
{
        LogicVector lv = new LogicVector(32);
        int mask = 1;
        for (int i = 0; i < 32; ++i, mask <<= 1)
                if ((mask & value) > 0)
                        lv.set(i);
        if (value < 0)
                lv.set(31);
        return lv;
}

public String toString(){
```

```
		String res = "";
		for(int i = size() - 1; i >= 0; i--){
			res += get(i)? "1" : "0";
		}
		return res + " (" + toInteger() + ")";
	}

	@Override
	public boolean equals(Object o){
		boolean res = false;
		if(o instanceof LogicVector){
			LogicVector v = (LogicVector)o;
			res = v.size() == this.size();
			if(res)
				for(int i = 0; i < size(); i++){
					res = v.get(i).booleanValue() == this.get(i).booleanValue();
					if(!res) break;
				}
		}else{
			throw new IllegalArgumentException("Argument must be a LogicVector instance.");
		}
		return res;
	}

	class HBoolean { boolean value = false; }

}
```

Archivo *qMIPS / qmips.presentation.builders.Builder*

```
public interface Builder {

	public void build();

	public Map<String, Device> getDisplayableDevices();

	public Clock getClock();

	public IMemory getInstrMemory();

	public Bus getResetBus();

	public ControlUnit getControlUnit();

	public SynchronousRegister programCounter();

}
```

Archivo *qMIPS / qmips.presentation.builders.QuantumMIPS*

```
public class QuantumMIPS implements Builder{


	private Map<String, Device> displayable;
	private Clock clockDev;
	private IMemory instrMemory;
	private Bus rst;
	private ControlUnit control;
	private SynchronousRegister programCounter;

	@Override
	public void build() {

		displayable = new TreeMap<String, Device>();

		//Buses:

		//Clock and reset
		Bus clk = new Bus(1);
		rst = new Bus(1);

		//Instruction pointer resolution
```

```
Bus instrPtr = new Bus(32);
Bus jmpAddr = new Bus(32);
Bus shftToJumpMux = new Bus(28);
Bus concatToMux = new Bus(32);
Bus andToOr = new Bus(1);
Bus xorToAnd = new Bus(1);

//Memory buses
Bus addr = new Bus(32);
Bus dataMem = new Bus(32);
Bus memDataToMux = new Bus(32);

//Instruction buses
Bus instr = new Bus(32);
Bus sExtToMuxAlu = new Bus(32);
Bus shftToMuxAlu = new Bus(32);

//Integer register file buses
Bus selW = new Bus(5);
Bus wrtData = new Bus(32);
Bus dataARegIn = new Bus(32);
Bus dataBRegIn = new Bus(32);
Bus dataARegOut = new Bus(32);
Bus dataBRegOut = new Bus(32);

//ALU buses
Bus aluDataA = new Bus(32);
Bus aluDataB = new Bus(32);
Bus aluOut = new Bus(32);
Bus aluOutHigh = new Bus(32);
Bus aluOutHighToMux = new Bus(32);
Bus aluFlags = new Bus(3);
Bus wbBus = new Bus(32);

//  Control buses  //
Bus pcWriteCond = new Bus(1);
Bus pcWrite = new Bus(1);
Bus iOrD = new Bus(1);
Bus memRead = new Bus(1);
Bus memWrite = new Bus(1);
Bus memToReg = new Bus(3);
Bus irWrite = new Bus(1);
Bus pcSource = new Bus(2);
Bus aluOp = new Bus(2);
Bus aluSrcB = new Bus(2);
Bus aluSrcA = new Bus(1);
Bus regWrite = new Bus(1);
Bus regDst = new Bus(2);
Bus solPCWrite = new Bus(1);
Bus aluControl = new Bus(4);
Bus aluHighWrite = new Bus(1);

Bus target = new Bus(1);
Bus qExe = new Bus(1);
//////////////////////

//Quantum buses
Bus mResult = new Bus(32);

//Devices:

//Memory
instrMemory = new AsyncMemory(dataBRegOut, dataMem, addr, memRead, memWrite, clk, 2048);

//Clock
clockDev = new Clock(clk);

//Program counter
programCounter = new SynchronousRegister(jmpAddr, instrPtr, solPCWrite, rst, clk);
displayable.put("Program counter", programCounter);

//Instruction register
displayable.put("Instruction register", new SynchronousRegister(dataMem, instr, irWrite, rst, clk));

//Memory data register
displayable.put("Memory data register", new SynchronousRegister(dataMem, memDataToMux, new Bus(1,1), rst, clk));

//A
```

```
                new SynchronousRegister(dataARegIn, dataARegOut, new Bus(1,1), rst, clk);

                //B
                new SynchronousRegister(dataBRegIn, dataBRegOut, new Bus(1,1), rst, clk);

                //ALU out
                new SynchronousRegister(aluOut, wbBus, new Bus(1,1), rst, clk);

                //ALU out high
                new SynchronousRegister(aluOutHigh, aluOutHighToMux, aluHighWrite, rst, clk);

                //Integer register file
                displayable.put("Register file", new RegisterFile(instr.getRange(21, 26), instr.getRange(16, 21), dataARegIn,
dataBRegIn, selW, regWrite, wrtData, rst, clk));

                //ALU
                new IntALU(aluDataA, aluDataB, aluControl, aluOut, aluOutHigh, aluFlags);

                //ALUControl
                new ALUControl(instr.getRange(0, 6), aluOp, aluControl);

                //Quantum unit
                QubitArray32 qa32 = new QubitArray32();

                new QubitTargetControl(instr.getRange(16, 21), instr.getRange(11, 16), target, clk, qa32);

                displayable.put("Quantum array state", new QuantumControl(instr.getRange(0, 6), instr.getRange(21, 26), dataARegOut,
mResult, qExe, clk, rst, qa32));

                //Multiplexers
                //1
                new Multiplexer(iOrD, new Bus[]{instrPtr, wbBus}, addr);

                //2
                new Multiplexer(regDst, new Bus[]{instr.getRange(16, 21), instr.getRange(11, 16), new Bus(31,5)}, selW);

                //3
                new Multiplexer(memToReg, new Bus[]{wbBus, memDataToMux, aluOutHighToMux, mResult, instrPtr}, wrtData);

                //4
                new Multiplexer(aluSrcA, new Bus[]{instrPtr, dataARegOut}, aluDataA);

                //5
                new Multiplexer(aluSrcB, new Bus[]{dataBRegOut,new Bus(4,32), sExtToMuxAlu, shftToMuxAlu}, aluDataB);

                //6
                new Multiplexer(pcSource, new Bus[]{aluOut, wbBus, concatToMux}, jmpAddr);

                //Sign extender
                new SignExt(instr.getRange(0, 16), sExtToMuxAlu);

                //Left shifters
                //1
                new ShiftLeft(sExtToMuxAlu, shftToMuxAlu, 2);

                //2
                new ShiftLeft(instr.getRange(0, 26), shftToJumpMux, 2);

                //Concatenator
                new Concat(new Bus[]{instrPtr.getRange(28, 32), shftToJumpMux}, concatToMux);

                //B Or gate
                new Logic(andToOr, pcWrite, solPCWrite, Logic.OR);

                //B And gate
                new Logic(pcWriteCond, xorToAnd, andToOr, Logic.AND);

                //B Xor gate
                new Logic(instr.getRange(26, 27), aluFlags.getRange(2, 3), xorToAnd, Logic.XOR);

                //Control unit
                control = new QuantumMIPSControlUnit(pcWriteCond, pcWrite, iOrD, memRead, memWrite, memToReg, irWrite, pcSource,
aluOp, aluSrcB, aluSrcA, regWrite, regDst, solPCWrite, aluControl, aluHighWrite, target, qExe, aluFlags.getRange(0, 1),
instr.getRange(26, 32), clk, rst);
                displayable.put("Control unit", (Device)control);

        }
```

```
	@Override
	public Map<String, Device> getDisplayableDevices() {
		return displayable;
	}


	@Override
	public Clock getClock() {
		return clockDev;
	}


	@Override
	public IMemory getInstrMemory() {
		return instrMemory;
	}


	@Override
	public Bus getResetBus() {
		return rst;
	}


	@Override
	public ControlUnit getControlUnit() {
		return control;
	}


	@Override
	public SynchronousRegister programCounter() {
		return programCounter;
	}

}
```

Archivo *qMIPS / qmips.sync.PoolSync*

```
/**
 *
 * Clase de sincronizacion de tareas.
 * Utiliza una piscina de hilos de tamaño indefinido, lo que
 * podria hacer dispararse el numero de hilos.
 * Las tareas se ejecutan por fases. Primero las activadas por el reloj,
 * cuando todas estas ha acabado se lanzan las que las anteriores hayan
 * activado y asi sucesivamente hasta que no despierte ninguna, momento
 * en que se libera al reloj.
 *
 * @author Jaime Coello de Portugal
 *
 */
public class PoolSync implements Synchronization {

	private ExecutorService pool;
	private Set<Runnable> waitingTasks;
	private int runningTasksNum = 1;

	public PoolSync() {
		waitingTasks = new HashSet<Runnable>();
		pool = Executors.newCachedThreadPool();
	}

	@Override
	public synchronized void activateProcesses(Set<Runnable> processes) {
		waitingTasks.addAll(processes);
	}

	@Override
	public synchronized void taskEnded() {
		runningTasksNum--;
		if (runningTasksNum <= 0) {
```

```
                        runningTasksNum = waitingTasks.size();
                        if (runningTasksNum == 0) {
                                notifyAll();
                        } else {
                                Set<Runnable> aux = waitingTasks;
                                waitingTasks = new HashSet<Runnable>();
                                for (Runnable r : aux) {
                                        try{
                                                pool.submit(r);
                                        }catch(RejectedExecutionException e){
                                                Log.inf.println("Execution canceled.");
                                        }
                                }
                        }
                }
        }

        @Override
        public synchronized void clockLockWait() throws InterruptedException{
                while (runningTasksNum != 0 || !waitingTasks.isEmpty()) {
                        wait();
                }
                runningTasksNum++;
        }

        @Override
        public void terminate() {
                try {
                        clockLockWait();
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
                pool.shutdown();
        }

        @Override
        public void exception(Exception e) {
                System.err.println(e.getMessage());
        }


}
```

Archivo *qMIPS / qmips.sync.Synchronization*

```
/**
 *
 * Esta interfaz define el comportamiento requerido a una clase
 * de sincronizacion, para poder definir varias.
 *
 * @author Jaime Coello de Portugal
 *
 */
public interface Synchronization {

        /**
         *
         * Lo invocan los buses cuando se se escribe sobre ellos para
         * pedir a la sincronizacion que se activen las tareas que le
         * tienen en su lista de sensibilidad.
         *
         * @param processes Las tareas que se deben activar.
         */
        public void activateProcesses(Set<Runnable> processes);

        /**
         *
         * Le indica a la sincronizacion que una tarea ha terminado.
         *
         */
        public void taskEnded();
```

```
	/**
	 *
	 * Invocado por el reloj para esperar en el monitor de la sincronizacion
	 * a que todas las tareas hayan concluido.
	 * @throws InterruptedException
	 *
	 */
	public void clockLockWait() throws InterruptedException;

	/**
	 *
	 * Indica a la sincronizacion que termine de operar y libere los recursos.
	 *
	 */
	public void terminate();

	/**
	 * Llamado en caso de excepcion en algun hilo de la piscina.
	 *
	 * @param e La excepcion producida.
	 */
	public void exception(Exception e);

}
```

Archivo *qMIPS / qmips.sync.SyncShortcut*

```
/**
 *
 * Acceso directo a la sincronizacion.
 * Aqui se define que sincronizacion se quiere usar.
 *
 * @author Jaime Coello de Portugal
 *
 */
public class SyncShortcut {

	public static Synchronization sync = new PoolSync();

}
```

A.2 Proyecto Qubit101

Archivo *Qubit101 / domain.engine.circuit.Circuit*

```
public class Circuit implements Iterable<Stage> {

	private Vector<Stage> circuit;
	private String name;

	public Circuit() {
		circuit = new Vector<Stage>();
	}

	public int getInputSize() {
		return circuit.get(0).getInputSize();
	}

	public int getOutputSize() {
		return circuit.lastElement().getOutputSize();
	}

	public QuantumState simulateCircuit(QuantumState input) {
		return simulateCircuit(input, 0);
	}

	public QuantumState simulateCircuit(QuantumState input, int disp) {
		return simulateCircuit(input, disp, new int[0]);
	}

	public QuantumState simulateCircuit(QuantumState input, int disp,
			int externalControlQubits[]) {
		QuantumState res = input;
		for (int i = 0; i < circuit.size() - 1; i++)
			if (circuit.get(i).getOutputSize() != circuit.get(i + 1)
					.getInputSize())
				throw new CircuitStructureException(
						"Different size between stage " + i
								+ " output and stage " + (i + 1) + " input.");

		for (Stage g : circuit) {
			res = g.simulate(res, disp, externalControlQubits);
		}

		return res;
	}

	public void addStage(Stage stage, int index) {
		circuit.add(index, stage);
	}

	public void removeStage(int index) {
		circuit.remove(index);
	}

	public int size() {
		return circuit.size();
	}

	public Stage getStage(int index) {
		return circuit.get(index);
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

	public String toString() {
		String res = "";
		for (Stage s : circuit) {
			res = res + s + "\n";
```

```
		}

		return res;
	}

	public Iterator<Stage> iterator() {
		return circuit.iterator();
	}

	class CircuitStructureException extends RuntimeException {

		private static final long serialVersionUID = 0x16757970492ed491L;

		public CircuitStructureException() {
			super();
		}

		public CircuitStructureException(String msg) {
			super(msg);
		}
	}

}
```

Archivo *Qubit101 / domain.engine.circuit.Stage*

```
public class Stage implements Iterable<Stage.Position> {

	public static final int GATE_HADAMARD = 1, GATE_SWAP = 2, GATE_PAULIX = 3,
			GATE_PAULIY = 4, GATE_PAULIZ = 5, GATE_PHASE = 6, GATE_CIRCUIT = 7;
	public static final int GATE_ADDQUBIT = -1, GATE_TRACE = -2,
			GATE_MEASURE = -3;
	public static final int CONTENT_EMPTY = 0, CONTENT_GATE = 1,
			CONTENT_CONTROLLED = 2, CONTENT_CONTROL = 3, CONTENT_FULL = 4;
	private Vector<Position> stage;

	public Stage(int inputStateSize) {
		stage = new Vector<Position>();
		for (int i = 0; i < inputStateSize; i++)
			stage.add(new Position(null, CONTENT_EMPTY));

	}

	public void addGate(Gate gate, int qubit) {
		if (gate.getGateId() == GATE_CIRCUIT) {
			Circuit c = ((CircuitGate) gate).getCircuit();
			if (qubit < 0 || qubit + c.getInputSize() > stage.size())
				throw new IndexOutOfBoundsException(
						"Circuit out of stage bounds.");
			for (int i = 0; i < c.getInputSize(); i++) {
				removeGate(qubit + i);
				if (i == 0)
					stage.set(qubit, new Position(gate, CONTENT_GATE));
				else
					stage.set(qubit + i, new Position(null, CONTENT_FULL));
			}

		} else {
			if (qubit < 0 || qubit >= stage.size())
				throw new IndexOutOfBoundsException("Gate out of stage bounds.");
			removeGate(qubit);
			stage.set(qubit, new Position(gate, CONTENT_GATE));
		}
	}

	public void addControlledGate(UnitaryGate gate, int qubit) {
		if (gate.getGateId() == GATE_CIRCUIT) {
			Circuit c = ((CircuitGate) gate).getCircuit();
			if (qubit < 0 || qubit + c.getInputSize() > stage.size())
				throw new IndexOutOfBoundsException(
						"Circuit out of stage bounds.");
			for (int i = 0; i < c.getInputSize(); i++) {
				removeGate(qubit + i);
```

```
                        if (i == 0)
                                stage.set(qubit, new Position(gate, CONTENT_CONTROLLED));
                        else
                                stage.set(qubit + i, new Position(null, CONTENT_FULL));
                }

        } else {
                if (qubit < 0 || qubit >= stage.size())
                        throw new IndexOutOfBoundsException("Gate out of stage bounds.");
                removeGate(qubit);
                stage.set(qubit, new Position(gate, CONTENT_CONTROLLED));
        }
}

public void addControlQubit(int qubit) {
        if (qubit < 0 || qubit >= stage.size()) {
                throw new IndexOutOfBoundsException("Gate out of stage bounds.");
        } else {
                removeGate(qubit);
                stage.set(qubit, new Position(null, CONTENT_CONTROL));
                return;
        }
}

public void removeGate(int qubit) {
        if (qubit < 0 || qubit >= stage.size())
                throw new IndexOutOfBoundsException("Index out of stage bounds.");
        Position old = (Position) stage.get(qubit);
        if (old.getContent() == CONTENT_FULL)
                removeGate(qubit - 1);
        else if ((old.getContent() == CONTENT_GATE || old.getContent() == CONTENT_CONTROLLED)
                        && old.getGate().getGateId() == GATE_CIRCUIT) {
                for (int i = 0; i < ((CircuitGate) old.getGate()).getCircuit()
                                .getInputSize(); i++)
                        stage.set(qubit + i, new Position(null, CONTENT_EMPTY));

        } else {
                stage.set(qubit, new Position(null, CONTENT_EMPTY));
        }
}

public void removeWire(int wire) {
        if (wire < 0 || wire >= stage.size()) {
                throw new IndexOutOfBoundsException("Index out of stage bounds.");
        } else {
                removeGate(wire);
                stage.remove(wire);
                return;
        }
}

public void addWire(int index) {
        if (index < 0 || index > stage.size())
                throw new IndexOutOfBoundsException("Index out of stage bounds.");
        if (index != stage.size()
                        && ((Position) stage.get(index)).getContent() == CONTENT_FULL)
                removeGate(index);
        stage.add(index, new Position(null, CONTENT_EMPTY));
}

public int getInputSize() {
        int res = 0;
        for (Position p : stage) {
                switch (p.getContent()) {
                case CONTENT_GATE:
                case CONTENT_CONTROLLED:
                        if (p.getGate().getGateId() != GATE_ADDQUBIT)
                                if(p.getGate().getGateId() == GATE_CIRCUIT)
                                        res = res + ((CircuitGate) p.getGate()).getCircuit().getInputSize();
                                else
                                        res++;
                        break;
                case CONTENT_FULL:
                        break;
                default:
                        res++;
                        break;
                }
```

```
		}

		return res;
	}

	public int getOutputSize() {
		int res = 0;
		for (Position p : stage) {
			switch (p.getContent()) {
			case CONTENT_GATE:
			case CONTENT_CONTROLLED:
				if (p.getGate().getGateId() != GATE_TRACE)
					if(p.getGate().getGateId() == GATE_CIRCUIT)
						res = res + ((CircuitGate) p.getGate()).getCircuit().getOutputSize();
					else
						res++;
				break;
			case CONTENT_FULL:
				break;
			default:
				res++;
				break;
			}
		}

		return res;
	}

	public int getInternalSize() {
		return stage.size();
	}

	public Position getPosition(int index) {
		return stage.get(index);
	}

	public QuantumState simulate(QuantumState input, int disp,
			int externalControlQubits[]) {
		QuantumState res = input;
		Vector<Integer> controlQubits = new Vector<Integer>();
		for (int i = 0; i < stage.size(); i++)
			if (stage.get(i).getContent() == CONTENT_CONTROL)
				controlQubits.add(Integer.valueOf(i + disp));
		int controlQubitsAuxArray[] = new int[controlQubits.size()];
		for (int i = 0; i < controlQubits.size(); i++)
			controlQubitsAuxArray[i] = ((Integer) controlQubits.get(i))
					.intValue();
		int controlQubitsArray[] = new int[controlQubits.size()
				+ externalControlQubits.length];
		System.arraycopy(controlQubitsAuxArray, 0, controlQubitsArray, 0,
				controlQubitsAuxArray.length);
		System.arraycopy(externalControlQubits, 0, controlQubitsArray,
				controlQubitsAuxArray.length, externalControlQubits.length);
		for (int i = 0; i < stage.size(); i++) {
			Position p = stage.get(i);
			switch (stage.get(i).getContent()) {
			default:
				break;
			case CONTENT_GATE:
				if (p.getGate().getGateId() > 0) {
					res = ((UnitaryGate) p.getGate()).operate(res, i + disp,
							externalControlQubits);
					break;
				}
				if (p.getGate().getGateId() >= 0)
					break;
				res = p.getGate().operate(res, i + disp);
				if (p.getGate().getGateId() == GATE_TRACE)
					disp--;
				break;
			case CONTENT_CONTROLLED:
				UnitaryGate g = (UnitaryGate) p.getGate();
				res = g.operate(res, i + disp, controlQubitsArray);
				break;
			}
		}

		return res;
```

```
	}

	public QuantumState simulate(QuantumState input, int disp) {
		return simulate(input, disp, new int[0]);
	}

	public QuantumState simulate(QuantumState input) {
		return simulate(input, 0);
	}

	public Iterator<Position> iterator() {
		return stage.iterator();
	}


	public class Position {

		public int getContent() {
			return content;
		}

		public Gate getGate() {
			return gate;
		}

		private Gate gate;
		private int content;

		public Position(Gate gate, int content) {
			super();
			this.gate = gate;
			this.content = content;
		}
	}

}
```

Archivo *Qubit101 / domain.engine.math.Complex*

```
public class Complex {

	private Double r;
	private Double i;

	/**
	 * Construct a new complex number.
	 * If the third argument is set to false,
	 * the complex number will be: a + b*i.
	 * If it is true, it will be: a*exp(i*b)
	 *
	 * @param a: real part if polar=false, else the magnitude of the complex vector.
	 * @param b: imaginary part if polar=false, else the argument of the complex vector.
	 * @param polar: true indicates a complex number in polar form.
	 */
	public Complex(Double a, Double b, Boolean polar){
		if(!polar){
			setReal(a);
			setImaginary(b);
		}else{
			setReal(a*Math.cos(b));
			setImaginary(i=a*Math.sin(b));
		}
	}

	/**
	 *
	 * Construct a new complex number.
	 * The complex number will be: a + b*i.
	 *
	 * @param a: real part.
	 * @param b: imaginary part.
	 */
	public Complex(Double a, Double b){
```

```
        this(a,b,false);
}

/**
 * Empty constructor, it creates a zero complex number.
 */
public Complex(){
        r=0.0;
        i=0.0;
}

/**
 * @return the real part of the complex number.
 */
public Double getReal(){
        return r;
}

/**
 * @return the imaginary part of the complex number.
 */
public Double getImaginary(){
        return i;
}

/**
 * @return the argument of the complex number in polar form.
 */
public Double getArgument(){
        Double res = 0.0;
        if(r.equals(0.0) && i.equals(0.0)){
                res=0.0;
        }else if(r>=0){
                res=Math.asin(i/getMagnitude());
        }else if(r<0){
                res=-Math.asin(i/getMagnitude())+Math.PI;
        }
        return res;
}

/**
 * @return the magnitude of the complex number.
 */
public Double getMagnitude(){
        return Math.sqrt(r*r+i*i);
}

/**
 * Set the real part of the complex number to r.
 * @param r: the real part of the complex number.
 */
public void setReal(Double r){
        this.r=r;
}

/**
 * Set the imaginary part of the complex number to i.
 * @param i: the imaginary part of the complex number.
 */
public void setImaginary(Double i){
        this.i=i;
}

/**
 * Set the argument part of the complex number to arg.
 * @param arg: the argument of the complex number.
 */
public void setArgument(Double arg){
        setReal(getMagnitude()*Math.cos(arg));
        setImaginary(getMagnitude()*Math.sin(arg));
}

/**
 * Set the magnitude of the complex number to mag.
 * @param mag: the magnitude of the complex number.
 */
public void setMagnitude(Double mag){
        setReal(mag*Math.cos(getArgument()));
```

```
                setImaginary(mag*Math.sin(getArgument()));
        }

        /**
         * Sums the complex number with the argument.
         * @param z: The complex number to be added.
         * @return The resulting complex number.
         */
        public Complex add(Complex z){
                return new Complex(r+z.getReal(),i+z.getImaginary(),false);
        }

        /**
         * Subtracts the complex number with the argument.
         * @param z: The complex number to be subtracted.
         * @return The resulting complex number.
         */
        public Complex subtract(Complex z){
                return new Complex(r-z.getReal(),i-z.getImaginary(),false);
        }

        /**
         * Multiplies the complex number with the argument.
         * @param z: The complex number to be multiplied.
         * @return The resulting complex number.
         */
        public Complex multiply(Complex z){
                return new Complex(r*z.getReal()-i*z.getImaginary(),i*z.getReal()+r*z.getImaginary(),false);
        }

        /**
         * Divides the complex number with the argument.
         * @param z: The complex divisor.
         * @return The resulting complex number.
         */
        public Complex divide(Complex z){
                if(z.equals(new Complex())) throw new ArithmeticException("Division by zero");
                return new
Complex((r*z.getReal()+i*z.getImaginary())/(z.getReal()*z.getReal()+z.getImaginary()*z.getImaginary()),(i*z.getReal()-
r*z.getImaginary())/(z.getReal()*z.getReal()+z.getImaginary()*z.getImaginary()),false);
        }

        /**
         * Returns the result of raising this complex number to the power
         * of the real argument.
         * @param exp: The exponent of the operation.
         * @return: The resulting complex number.
         */
        public Complex pow(Double exp){
                return new Complex(Math.pow(getMagnitude(), exp),getArgument()*exp,true);
        }

        /**
         * @return the complex conjugate of this complex number.
         */
        public Complex conjugate(){
                return new Complex(getReal(),-getImaginary(),false);
        }




        public boolean equals(Object o){
                if(!(o instanceof Complex)) throw new IllegalArgumentException("The object must be a complex number");
                Complex z = (Complex)o;
                return(z.getReal().equals(r) && z.getImaginary().equals(i));
        }

        public String toString(){
                String res="";
                double dr = new BigDecimal(r).setScale(15, BigDecimal.ROUND_HALF_DOWN).doubleValue();
                double di = new BigDecimal(i).setScale(15, BigDecimal.ROUND_HALF_DOWN).doubleValue();
                if(dr==0.0 && di==0.0){
                        res="0.0";
                }else if(dr==0.0){
                        res=di+"i";
                }else if(di==0.0){
                        res=String.valueOf(dr);
```

```
                }else{
                        res = dr + "+" + di + "i";
                }
                return res;
        }

        /**
         * @return the polar form representation of this complex number.
         */
        public String polarForm(){
                String res = "";
                Double m = getMagnitude();
                Double a = getArgument();
                if(a == 0.0){
                        res = String.valueOf(m);
                }else if(m==0.0){
                        res = "0.0";
                }else if(m==1.0){
                        res = "exp(" + a +"*i)";
                }else{
                        res = m + "*exp(" + a +"*i)";
                }
                return res;
        }
}
```

Archivo *Qubit101 domain.engine.quantum.gates.AddQubit*

```
public class AddQubit implements Gate {

        public QuantumState operate(QuantumState input, int targetQubit) {
                QuantumState res = new QuantumState();
                byte stateRes[];
                for (Entry<ClassicState, Complex> e : input) {
                        byte state[] = e.getKey().getState();
                        stateRes = new byte[state.length + 1];
                        int j = 0;
                        for (int i = 0; i < stateRes.length; i++)
                                if (i != targetQubit) {
                                        stateRes[i] = state[j];
                                        j++;
                                } else {
                                        stateRes[i] = 0;
                                }
                        res.add(e.getValue(), new ClassicState(stateRes));
                }

                return res;
        }

        public int getGateId() {
                return Stage.GATE_ADDQUBIT;
        }

        public String getSymbol() {
                return "|0>";
        }
}
```

Archivo *Qubit101 / domain.engine.quantum.gates.CircuitGate*

```
public class CircuitGate
    implements UnitaryGate
{

    public CircuitGate(Circuit circuit)
    {
        setCircuit(circuit);
    }

    public QuantumState operate(QuantumState input, int targetQubit)
```

```
    {
        return circuit.simulateCircuit(input, targetQubit);
    }

    public int getGateId()
    {
        return 7;
    }

    public String getSymbol()
    {
        return circuit.getName();
    }

    public QuantumState operate(QuantumState input, int targetQubit, int controlQubits[])
    {
        return circuit.simulateCircuit(input, targetQubit, controlQubits);
    }

    public Circuit getCircuit()
    {
        return circuit;
    }

    public void setCircuit(Circuit circuit)
    {
        this.circuit = circuit;
    }

    private Circuit circuit;
}
```

Archivo *Qubit101 / domain.engine.quantum.gates.HadamardGate*

```
public class HadamardGate extends UnitaryGateTemplate {

        private final double SQRT05 = 0.70710678118654757;
        private final Complex factor = new Complex(SQRT05, 0.0, false);

        public int getGateId() {
                return Stage.GATE_HADAMARD;
        }

        public void singleComponentOperation(Entry<ClassicState, Complex> e,
                        int targetQubit, QuantumState res) {
                byte state0[] = (byte[]) ((ClassicState) e.getKey()).getState().clone();
                byte state1[] = (byte[]) state0.clone();
                Complex phase = state0[targetQubit] != 1 ? new Complex(1.0, 0.0,
                                Boolean.valueOf(false)) : new Complex(-1.0, 0.0, false);
                state0[targetQubit] = 0;
                state1[targetQubit] = 1;
                res.add(e.getValue().multiply(factor), new ClassicState(state0));
                res.add(e.getValue().multiply(factor).multiply(phase),
                                new ClassicState(state1));
        }

        public String getSymbol() {
                return "H";
        }

}
```

Archivo *Qubit101 / domain.engine.quantum.gates.Measure*

```
public class Measure implements Gate {

        private byte measurement;

        public Measure() {
```

```
                measurement = -1;
        }

        public QuantumState operate(QuantumState input, int targetQubit) {
                QuantumState res = new QuantumState();
                double random = (new Random()).nextDouble();
                double prob0 = 0.0D;
                for (Entry<ClassicState, Complex> e : input) {
                        if (e.getKey().getState()[targetQubit] == 0)
                                prob0 += Math.pow(e.getValue().getMagnitude().doubleValue(),
                                                2.0);
                }

                measurement = 0;
                if (random > prob0)
                        measurement = 1;
                if (prob0 == 0.0D) {
                        res = input;
                } else {
                        Complex norm[] = {
                                        new Complex(Double.valueOf(1.0D / Math.sqrt(prob0)),
                                                        Double.valueOf(0.0D), Boolean.valueOf(false)),
                                        new Complex(Double.valueOf(1.0D / Math.sqrt(1.0D - prob0)),
                                                        Double.valueOf(0.0D), Boolean.valueOf(false)) };
                        for (Entry<ClassicState, Complex> e : input) {
                                if (e.getKey().getState()[targetQubit] == measurement)
                                        res.add(e.getValue().multiply(norm[measurement]),
                                                        (ClassicState) e.getKey().clone());
                        }

                }
                return res;
        }

        public byte getMeasurementResult() {
                if (measurement == -1)
                        throw new IllegalStateException("Measurement still not computed.");
                else
                        return measurement;
        }

        public int getGateId() {
                return -3;
        }

        public String getSymbol() {
                return "Meas";
        }

}
```

Archivo *Qubit101 / domain.engine.quantum.gates.PauliXGate*

```
public class PauliXGate extends UnitaryGateTemplate
{

    public int getGateId()
    {
        return Stage.GATE_PAULIX;
    }

    public void singleComponentOperation(Entry<ClassicState, Complex> e, int targetQubit, QuantumState res)
    {
        byte state[] = e.getKey().getState();
        byte stateneg[] = (byte[])state.clone();
        stateneg[targetQubit] = (byte)(stateneg[targetQubit] != 0 ? 0 : 1);
        res.add(e.getValue(), new ClassicState(stateneg));
    }

    public String getSymbol()
    {
        return "X";
    }
```

```
}
```

Archivo *Qubit101 / domain.engine.quantum.gates.PauliYGate*

```
public class PauliYGate extends UnitaryGateTemplate {

	public int getGateId() {
		return Stage.GATE_PAULIY;
	}

	public void singleComponentOperation(Entry<ClassicState, Complex> e,
			int targetQubit, QuantumState res) {
		byte state[] = e.getKey().getState();
		byte stateRes[] = (byte[]) state.clone();
		Complex phase = null;
		if (stateRes[targetQubit] == 0) {
			stateRes[targetQubit] = 1;
			phase = e.getValue().multiply(new Complex(0.0, -1.0, false));
		} else {
			stateRes[targetQubit] = 0;
			phase = e.getValue().multiply(new Complex(0.0D, 1.0D, false));
		}
		res.add(phase, new ClassicState(stateRes));
	}

	public String getSymbol() {
		return "Y";
	}
}
```

Archivo *Qubit101 / domain.engine.quantum.gates.PauliZGate*

```
public class PauliZGate extends UnitaryGateTemplate
{

    public int getGateId()
    {
        return Stage.GATE_PAULIZ;
    }

    public void singleComponentOperation(Entry<ClassicState, Complex> e, int targetQubit, QuantumState res)
    {
        byte state[] = e.getKey().getState();
        byte stateRes[] = (byte[])state.clone();
        if(stateRes[targetQubit] == 1)
            res.add(e.getValue().multiply(new Complex(-1.0, 0.0, false)), new ClassicState(stateRes));
        else
            res.add(e.getValue(), new ClassicState(stateRes));
    }

    public String getSymbol()
    {
        return "Z";
    }
}
```

Archivo *Qubit101 / domain.engine.quantum.gates.PhaseShiftGate*

```
public class PhaseShiftGate extends UnitaryGateTemplate
{

	private String alfa;

    public PhaseShiftGate(String alfa)
    {
        setAlfa(alfa);
    }

    public int getGateId()
```

```
    {
        return Stage.GATE_PHASE;
    }

    public void singleComponentOperation(Entry<ClassicState, Complex> e, int targetQubit, QuantumState res)
    {
        byte state[] = e.getKey().getState();
        byte stateRes[] = (byte[])state.clone();
        JEP j = new JEP();
                j.addStandardConstants();
                j.addStandardFunctions();
                j.setImplicitMul(true);
                double value = 0.0;
                try {
                        value = (Double) j.evaluate(j.parse(alfa));
                } catch (ParseException e1) {
                        System.out.println("Error parsing phase gate equation.");
                }
        if(stateRes[targetQubit] == 1)
            res.add(e.getValue().multiply(new Complex(1.0, value, true)), new ClassicState(stateRes));
        else
            res.add(e.getValue(), new ClassicState(stateRes));
    }

    public void setAlfa(String alfa)
    {
        this.alfa = alfa;
    }

    public String getAlfa()
    {
        return alfa;
    }

    public String getSymbol()
    {
        return "P( " + alfa + " )";
    }


}
```

Archivo *Qubit101 / domain.engine.quantum.gates.TraceOut*

```
public class TraceOut implements Gate {

        public QuantumState operate(QuantumState input, int targetQubit) {
                QuantumState res = new QuantumState();
                QuantumState input2 = new Measure().operate(input, targetQubit);
                byte stateRes[];
                for (Entry<ClassicState, Complex> e : input2) {
                        byte state[] = e.getKey().getState();
                        stateRes = new byte[state.length - 1];
                        int j = 0;
                        for (int i = 0; i < state.length; i++)
                                if (i != targetQubit) {
                                        stateRes[j] = state[i];
                                        j++;
                                }
                        res.add(e.getValue(), new ClassicState(stateRes));
                }
                return res;
        }

        public int getGateId() {
                return Stage.GATE_TRACE;
        }

        public String getSymbol() {
                return "Tr";
        }
}
```

Archivo *Qubit101 / domain.engine.quantum.gates.UnitaryGateTemplate*

```
public abstract class UnitaryGateTemplate implements UnitaryGate {

	public QuantumState operate(QuantumState input, int targetQubit) {
		QuantumState res = new QuantumState();
		for (Entry<ClassicState, Complex> e : input)
			singleComponentOperation(e, targetQubit, res);

		return res;
	}

	public abstract int getGateId();

	public QuantumState operate(QuantumState input, int targetQubit,
			int controlQubits[]) {
		QuantumState res = new QuantumState();
		for (Entry<ClassicState, Complex> e : input) {
			byte state0[] = e.getKey().getState();
			boolean act = true;
			for (int i = 0; i < controlQubits.length; i++) {
				if (state0[controlQubits[i]] != 1){
					act = false;
					break;
				}
			}
			if (act)
				singleComponentOperation(e, targetQubit, res);
			else
				res.add(e.getValue(), e.getKey());
		}

		return res;
	}

	public abstract void singleComponentOperation(
			Entry<ClassicState, Complex> entry, int i, QuantumState quantumstate);

	public abstract String getSymbol();
}
```

Archivo *Qubit101 / domain.engine.quantum.interfaces.Gate*

```
public interface UnitaryGate extends Gate {

	public abstract QuantumState operate(QuantumState quantumstate, int i,
			int ai[]);
}
```

Archivo *Qubit101 / domain.engine.quantum.maps.IterableMap*

```
public interface IterableMap extends Map<ClassicState, Complex>, Iterable<Entry<ClassicState, Complex>>{}
```

Archivo *Qubit101 / domain.engine.quantum.maps.IterableTreeMap*

```
public class IterableTreeMap extends TreeMap<ClassicState, Complex> implements IterableMap{

	private static final long serialVersionUID = -3742358892070112142L;

	@Override
	public Iterator<java.util.Map.Entry<ClassicState, Complex>> iterator() {
		return this.entrySet().iterator();
	}

}
```

Archivo *Qubit101 / domain.engine.quantum.ClassicState*

```
public class ClassicState implements Comparable<ClassicState> {

	private byte state[];

	public ClassicState(byte state[]) {
		this.state = state;
	}

	public byte[] getState() {
		return state;
	}

	public int compareTo(ClassicState o) {
		int res = 0;
		for (int i = 0; res == 0 && i < state.length; i++)
			res = state[i] - o.state[i];

		return res;
	}

	public boolean equals(ClassicState o) {
		boolean res = true;
		for (int i = 0; i < state.length; i++) {
			if (o.state[i] == state[i])
				continue;
			res = false;
			break;
		}

		return res;
	}

	public String toString() {
		String res = "";
		for (int i = 0; i < state.length; i++)
			res = res + state[i];

		return res;
	}

	public Object clone() {
		ClassicState res = new ClassicState(new byte[state.length]);
		for (int i = 0; i < state.length; i++)
			res.state[i] = state[i];

		return res;
	}


}
```

Archivo *Qubit101 / domain.engine.quantum.ClassicState*

```
public class QuantumState implements Iterable<Entry<ClassicState, Complex>> {

	private static final Complex ZERO = new Complex();

	public static final int TREEMAPTYPE = 1, ARRAYSORTEDMAPTYPE = 2,
			NATIVETREEMAPTYPE = 3, NATIVEARRAYMAPTYPE = 4;
	private IterableMap qState;

	public QuantumState() {
		qState = new IterableTreeMap();
	}

	public QuantumState(int mapType){
		switch(mapType){
		case TREEMAPTYPE:
			qState = new IterableTreeMap();
			break;
		case ARRAYSORTEDMAPTYPE:
			qState = new ArraySortedMap();
			break;
```

```
                case NATIVETREEMAPTYPE:
                        qState = new NativeMap();
                        break;
                case NATIVEARRAYMAPTYPE:
                        qState = new NativeArrayMap();
                        break;
                default:
                        throw new IllegalArgumentException("Unknown map type.");
                }
        }

        public void add(Complex coef, ClassicState cs) {
                Complex tmp = qState.get(cs);
                if (tmp != null) {
                        tmp = tmp.add(coef);
                        qState.put(cs, tmp);
                        if (tmp.equals(ZERO))
                                qState.remove(cs);
                } else if (!coef.equals(ZERO))
                        qState.put(cs, coef);
        }

        public String toString() {
                Iterator<Entry<ClassicState, Complex>> i = qState.iterator();
                String res;
                Entry<ClassicState, Complex> ent;
                ClassicState c;
                for (res = ""; i.hasNext(); res = res+ent.getValue()+"|"+c+"> + \n") {
                        ent = i.next();
                        c = (ClassicState) ent.getKey();
                }
                if (res.length() > 0)
                        return res.substring(0, res.length() - 3);
                else
                        return "0.0";
        }

        public Iterator<Entry<ClassicState, Complex>> iterator() {
                return qState.iterator();
        }

        public Iterator<String> stringIterator() {
                return new StringIterator();
        }

        private class StringIterator implements Iterator<String> {

                Iterator<Entry<ClassicState, Complex>> it = qState.entrySet().iterator();
                int length = qState.size();
                int cont = 0;

                public String next() {
                        Entry<ClassicState, Complex> e = it.next();
                        cont++;
                        String res;
                        if (cont == length)
                                res = e.getValue() + " |" + e.getKey() + ">";
                        else
                                res = e.getValue() + " |" + e.getKey() + "> + ";
                        return res;
                }

                public boolean hasNext() {
                        return it.hasNext();
                }

                public void remove() {}

        }

}
```

Archivo *Qubit101 / domain.files.CircuitLoader*

```
public class CircuitLoader extends XMLLoader{
        private Circuit circuit;
        private String circuitName;

        public CircuitLoader(File source) throws ParserConfigurationException, SAXException, IOException{
                 super(source);
                 buildCircuit();
        }

        public CircuitLoader(Node n) throws ParserConfigurationException{
                 super(n);
                 buildCircuit();
        }


        private void buildCircuit() throws ParserConfigurationException{
                 Document DOM = getDOM();
                 Node NCir = DOM.getChildNodes().item(0);
                 Map<String,String> atNCir = getAttributes(NCir);
                 int size=Integer.valueOf(atNCir.get("size"));
                 circuitName = atNCir.get("name");
                 circuit = new Circuit();
                 circuit.setName(circuitName);
                 int x = 0;
                 for(int i = 0; x < size; i++){
                          Node n = NCir.getChildNodes().item(i);
                          if(n.getNodeName().equals("stage")){
                     buildStage(n, x);
                     x++;
                 }
                      }
        }

    private void buildStage(Node n, int index) throws ParserConfigurationException{
        NodeList nl = n.getChildNodes();
        Stage stage = new Stage(Integer.parseInt(getAttributes(n).get("size")));
        circuit.addStage(stage, index);
        for(int i = 0; i< n.getChildNodes().getLength(); i++){
            Node nc = nl.item(i);
            if(nc.getNodeName().equals("gate")){
                Map<String,String> att = getAttributes(nc);
                int qubit = Integer.valueOf(att.get("qubit"));
                int gateId = Integer.valueOf(att.get("gate_id"));
                if(att.containsKey("controlled") && att.get("controlled").equals("true"))
                    stage.addControlledGate((UnitaryGate) getGateFromId(gateId,nc), qubit);
                else
                 stage.addGate(getGateFromId(gateId,nc), qubit);
            }else if(nc.getNodeName().equals("control")){
                Map<String,String> att = getAttributes(nc);
                int qubit = Integer.valueOf(att.get("qubit"));
                stage.addControlQubit(qubit);
            }else if(nc.getNodeName().equals("qcircuit")){
                Map<String,String> att = getAttributes(nc);
                int qubit = Integer.valueOf(att.get("qubit"));
                boolean controlled = "true".equals(att.get("controlled"));
                Circuit innc = buildCircuitGate(nc);
                if(!controlled)
                 stage.addGate(new CircuitGate(innc),qubit);
                else
                 stage.addControlledGate(new CircuitGate(innc),qubit);
            }
         }
    }

        private Circuit buildCircuitGate(Node n) throws ParserConfigurationException{
                 CircuitLoader cl = new CircuitLoader(n);
                 return cl.circuit;
        }

    public Circuit getCircuit(){
        return circuit;
    }

    public String getCircuitName(){
        return circuitName;
    }

    private Gate getGateFromId(int id, Node nc){
```

```
        Gate g = null;
        switch(id){
        case Stage.GATE_ADDQUBIT:
                 g = new AddQubit();
                 break;
        case Stage.GATE_HADAMARD:
                 g = new HadamardGate();
                 break;
        case Stage.GATE_MEASURE:
                 g = new Measure();
                 break;
        case Stage.GATE_PAULIX:
                 g = new PauliXGate();
                 break;
        case Stage.GATE_PAULIY:
                 g = new PauliYGate();
                 break;
        case Stage.GATE_PAULIZ:
                 g = new PauliZGate();
                 break;
        case Stage.GATE_PHASE:
                 g = new PhaseShiftGate(getAttributes(nc).get("alpha"));
                 break;
        case Stage.GATE_TRACE:
                 g = new TraceOut();
                 break;
        }
        return g;
    }

}
```

Archivo *Qubit101 / domain.files.CircuitSaver*

```
public class CircuitSaver extends XMLSaver {

    private static Document dom;

    private CircuitSaver(Document d, File f) throws FileNotFoundException, XMLStreamException, IOException {
        super(d, f);
    }

    public CircuitSaver(Circuit c, File f) throws FileNotFoundException, XMLStreamException, ParserConfigurationException,
IOException {
        super(buildDocument(c, f.getName()), f);
    }

    private static Document buildDocument(Circuit c, String name) throws ParserConfigurationException {
        dom = buildDOM();
        dom.appendChild(buildNodeTree(c, name, null, null));
        dom.normalize();
        return dom;
    }

    private static Node buildNodeTree(Circuit c, String name, String qubit, String controlled) throws ParserConfigurationException {
        Element nCir = dom.createElement("qcircuit");
        nCir.setAttribute("name", name);
        nCir.setAttribute("size", String.valueOf(c.size()));
        if(qubit != null)
                 nCir.setAttribute("qubit", qubit);
        if(controlled != null)
                 nCir.setAttribute("controlled", controlled);
        for (int j = 0; j < c.size(); j++) {
            generateStageDOM(nCir, c.getStage(j));
        }
        return nCir;
    }

    private static void generateStageDOM(Element e, Stage stage) throws DOMException, ParserConfigurationException {
        Element eSt = dom.createElement("stage");
        eSt.setAttribute("size", Integer.toString(stage.getInternalSize()));
        e.appendChild(eSt);
        for (int j = 0; j < stage.getInternalSize(); j++) {
```

```
            Position p = stage.getPosition(j);
            Element eG;
            switch (p.getContent()) {
                case Stage.CONTENT_GATE:
                case Stage.CONTENT_CONTROLLED:
                 if(p.getGate().getGateId() == Stage.GATE_CIRCUIT){
                          Circuit cir = ((CircuitGate)p.getGate()).getCircuit();
                        eG = dom.createElement("measure");
                        eG.setAttribute("qubit", Integer.toString(j));
                        if(p.getContent() == Stage.CONTENT_CONTROLLED)
                          eSt.appendChild(buildNodeTree(cir, cir.getName(),Integer.toString(j), "true"));
                        else
                          eSt.appendChild(buildNodeTree(cir, cir.getName(),Integer.toString(j), null));
                 }else{
                          eG = dom.createElement("gate");
                        eG.setAttribute("gate_id", Integer.toString(p.getGate().getGateId()));
                        eG.setAttribute("qubit", Integer.toString(j));
                        if(p.getContent() == Stage.CONTENT_CONTROLLED)
                          eG.setAttribute("controlled", "true");
                        if(p.getGate().getGateId() == Stage.GATE_PHASE)
                          eG.setAttribute("alpha",((PhaseShiftGate)p.getGate()).getAlfa());
                        eSt.appendChild(eG);
                 }
                    break;
                case Stage.CONTENT_CONTROL:
                    eG = dom.createElement("control");
                    eG.setAttribute("qubit", Integer.toString(j));
                    eSt.appendChild(eG);
                    break;
            }
        }
    }

    private static Document buildDOM() {
        Document d = null;
        try {
            d = (DocumentBuilderFactory.newInstance().newDocumentBuilder()).newDocument();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return d;
    }
}
```

Archivo *Qubit101 / domain.files.XMLLoader*

```
public class XMLLoader {
        private File source;
        private Document DOM;

        public XMLLoader(File source) throws ParserConfigurationException, SAXException, IOException{
                  if(!source.exists()){
                            throw new FileNotFoundException();
                  }
                  this.source=source;
                  DocumentBuilderFactory df =DocumentBuilderFactory.newInstance();
                  DocumentBuilder d = df.newDocumentBuilder();
                  DOM = d.parse(source);
        }

        public XMLLoader(Node n) throws ParserConfigurationException{
                  DOM = nodeToDOM(n);
        }

        public Document getDOM(){
                  return DOM;
        }

        public Document nodeToDOM(Node n) throws ParserConfigurationException{
                  DocumentBuilderFactory df =DocumentBuilderFactory.newInstance();
                  DocumentBuilder d = df.newDocumentBuilder();
                  Document dom = d.newDocument();
                  dom.appendChild(dom.importNode(n, true));
```

```
                return dom;
        }

        public File getSource(){
                return source;
        }

        public Map<String,String> getAttributes(Node n){
                Map<String,String> m = new TreeMap<String,String>();
                for(int i=0;i<n.getAttributes().getLength();i++){
                        Node e = n.getAttributes().item(i);
                        m.put(e.getNodeName(), e.getNodeValue());
                }
                return m;
        }

}
```

Archivo *Qubit101 / domain.files.XMLSaver*

```
public class XMLSaver {
        XMLStreamWriter xt;
    private boolean firstLine = true;

        public XMLSaver(Document d, File f) throws FileNotFoundException, XMLStreamException, IOException{
                XMLOutputFactory xof = XMLOutputFactory.newInstance();
         File fext = null;
         if(getExtension(f).equalsIgnoreCase(".qcml")){
             fext=f;
         }else
             fext = new File(f.getAbsolutePath()+".qcml");
         xt = xof.createXMLStreamWriter(new FileWriter(fext));
                generateXML(d.getFirstChild(),"");
         xt.flush();
                xt.close();
        }

        private void generateXML(Node n, String desp) throws XMLStreamException{
         boolean empty = n.getChildNodes().getLength()==0;
         if(firstLine){
              xt.writeCharacters(desp);
              firstLine=false;
         }
         else
             xt.writeCharacters("\n" + desp);
         if(empty)
             xt.writeEmptyElement(n.getNodeName());
         else
             xt.writeStartElement(n.getNodeName());
                Map<String, String> at = getAttributes(n);
                Iterator<Map.Entry<String, String>> it = at.entrySet().iterator();
                while(it.hasNext()){
                        Map.Entry<String, String> s = it.next();
                        xt.writeAttribute(s.getKey(), s.getValue());
                }
                for(int i=0;i<n.getChildNodes().getLength();i++){
                        if(n.getChildNodes().item(i).getNodeName().equalsIgnoreCase("#text")){
                                xt.writeCharacters(n.getChildNodes().item(i).getTextContent());
                        }else
                                generateXML(n.getChildNodes().item(i),desp+"\t");
                }
         if(!(n.getChildNodes().getLength()==1 && n.getFirstChild().getNodeType()==Node.TEXT_NODE) && !empty)
            xt.writeCharacters("\n"+desp);
         if(!empty)
             xt.writeEndElement();
        }

        private Map<String,String> getAttributes(Node n){
                Map<String,String> m = new TreeMap<String,String>();
                for(int i=0;i<n.getAttributes().getLength();i++){
                        Node e = n.getAttributes().item(i);
                        m.put(e.getNodeName(), e.getNodeValue());
                }
```

```
                return m;
        }

    private String getExtension(File f){
        String name = f.getName();
        if(name.lastIndexOf(".")==-1){
            return "";
        }else
            return name.substring(name.lastIndexOf("."),name.length());
    }
}
```